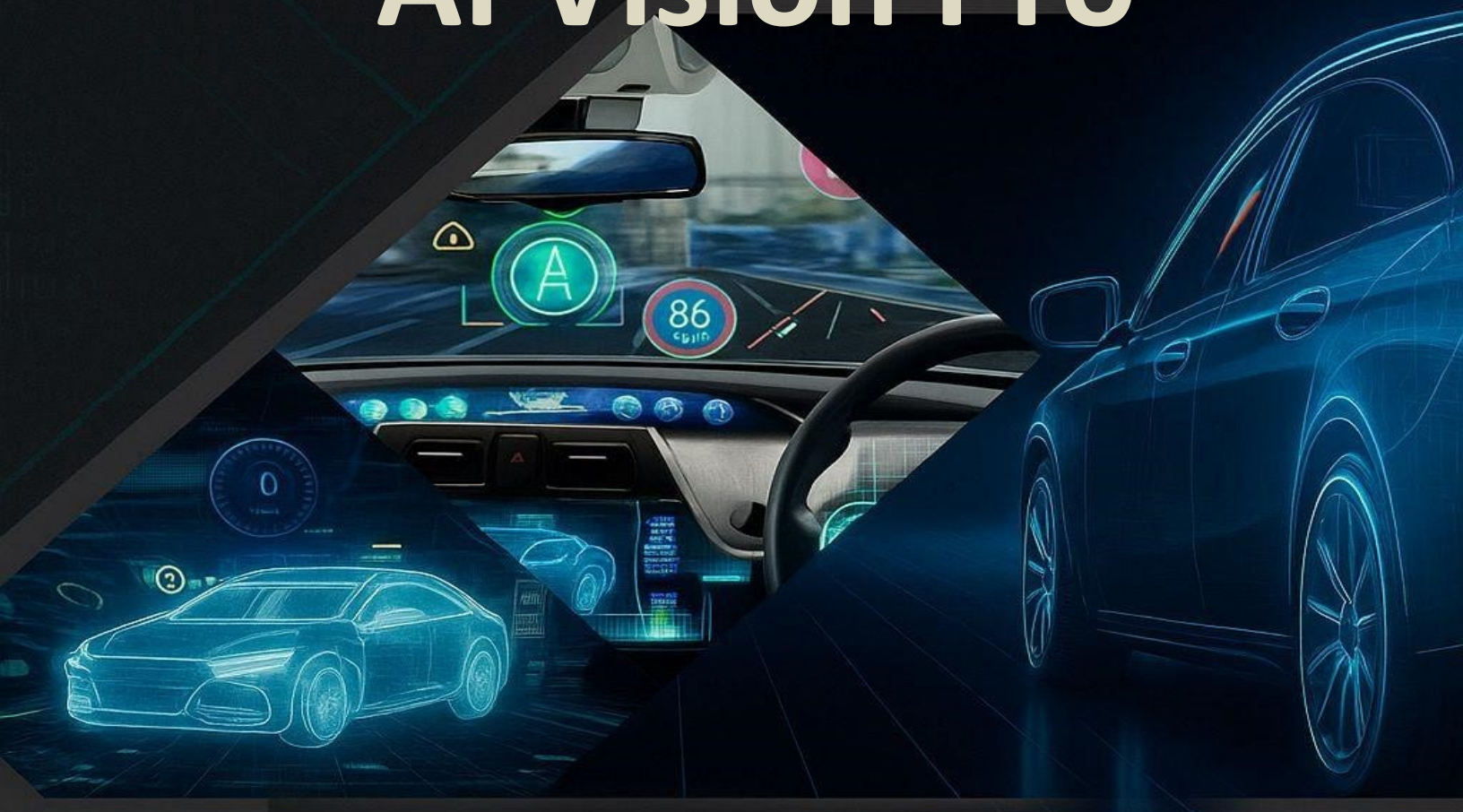
AL-Azhar University
Faculty of Engineering
Electronics & Communication Department

Supervised by:
Dr. Mohamed Yasin

Logi Drive
AI Vision Pro



Represented by:

Omar Mohammed Youssef 104511
Mohammed Refaat Hamdallah 104514
Ammar Ashraf Mohammed 104510
Ali Akram Abdelzahra 104527
Moawia Mohammed Abdelsabour 104524

Acknowledgement

Our primary thanks go to **Allah** for the Success in the Completion of this work.
Foremost, We would like to express our sincere gratitude to our supervisor, **Dr. Mohamed Yasin**, for the continuous support of our graduation project and research, and for his patience, motivation, and immense knowledge. We could not have imagined having a better advisor and mentor for our graduation project.
Also, we cannot forget our college **Faculty of Engineering Al-Azhar University** for giving us a good education, help and chances to achieve our goals.
Finally, we will never find enough words to express the gratitude and appreciation that we owe to our families for their constant support, encouragement, never- ending patience as well as their love that raised us up again when we got weary and without them, we doubt that we would be in this place today.

Abstract

Modern vehicles rely extensively on advanced electronic systems designed to improve safety, comfort, and performance. As these systems become more complex, the potential for critical system failures that may not be immediately apparent to the driver also increases. Drivers frequently depend on these systems for operational efficiency and safety, and malfunctions are sometimes only detected after they have occurred. Such failures may result in unsafe conditions, increased repair expenses, and disruptions to regular vehicle use.

From 2010 to 2020, over 100 million vehicle recalls were issued in the United States, with a notable portion involving software-related problems.

2018: Reports indicated that nearly half of automotive recalls in this year were attributed to software defects.

2019: Software issues accounted for approximately 15% of all automotive recalls in the U.S., impacting millions of vehicles.

To address these concerns, this project involves the development of a smart car equipped with a logger system to monitor and verify the operation of all critical vehicle systems prior to each journey. The logger is intended to identify and notify the driver of any potential issues in real time, thereby aiming to reduce the likelihood of unforeseen system failures while driving. In addition, the car utilizes FOTA (Firmware Over-The-Air) technology to enable remote software updates and consistent communication with service centers, supporting improved reliability and system performance.

By continuously updating the driver about the vehicle's condition, this system seeks to support timely decision-making and minimize the occurrence of unexpected breakdowns, with the objective of improving road safety and managing maintenance costs.[1]

Content

Acknowledgement.....
Abstract .....
Content .....
List of figures .....
INTRODUCTION.....
0.1 How does our project solve these problems?....., 18
0.2 Project Descriptions .....
0.3 Project Objectives .....
0.4 Expected Outcomes……20
0.5 Research Methodology, Implementation and Validation: .....
0.6 Project Architecture .....
Module A .....
CHAPTER 1: BLUE PILL(STM32F103C8T6) .....
1.1 Introduction.....
1.2 Why STM32f103c8t6 ?.....
1.3 STM32F103C8T6 Board Features.....
1.4 STM32F103C8T6 Board Hardware .....
1.4.1 Hardware component of STM32F103C8T6.....
1.4.2 Layout the GPIO Headers.....
1.5 STM32F103C8T6 OS .....
CHAPTER 2 : RASPBERRYPI .....
2.1 Introduction.....
2.2 Why Raspberry Pi? .....
2.3 Why Raspberry Pi Versions? .....
2.4 Raspberry Pi Hardware .....
2.4.1 Specifications Pin Diagram and Description .....
2.4.2 Hardware Components of Raspberry Pi .....
2.4.3 Layout of The GPIO Header.....
2.5 Raspberry Pi OS.....
2.6 Uses of Raspberry Pi.....
2.7 Programming the Raspberry Pi.....
CHAPTER 3:SENSORS & ACTUATORS.....

3.1 INFRARED SENSOR [8]
3.1.1 Introduction
3.1.2 About the Module
3.2 Buzzer
3.2.1 Introduction
3.2.2 Buzzer Pin Configuration
3.2.3 Specifications
3.2.4 Working principle
3.3 DC Motor
3.3.1 Introduction
3.3.2 Basic Principles and Operation
3.3.3 Types of DC Motors
3.3.4 Detailed Components and Design
3.3.5 Performance Characteristics
3.4 H-bridge
3.4.1 Introduction
3.4.2 Basic Principles
3.4.3 Components of an H-Bridge
3.4.4 Operation
3.5 Infrared Night Vision
3.5.1 Introduction
3.5.2 About the Module
3.5.3 Specifications
3.5.4 How the Night Vision Camera Works
3.5.5 Night Vision Camera Usage in Our Project
3.6 Bluetooth Module HC-05
3.6.1 Introduction
3.6.2 About the Module
3.6.3 Specifications
3.6.4 How the Bluetooth Module Works
CHAPTER 4: The Design
4.1 Introduction
4.2 Design Program Used
4.3 The Design Overview

Module B
CHAPTER 5: Controller Area Network (CAN)
5.1 Introduction
5.2 What is CAN?
5.3 Features of CAN protocol
5.4 Can Network Message Format
5.5 Message frame
5.6 CAN Protocol Working Principle
5.7 Operation of the CAN Network
5.8 Interface between Nodes and Communication Network
5.9 CAN in Action
CHAPTER 6: D-Bus
6.1 Introduction to D-bus
6.2 D-BUS Concepts Service Object Interface service:
6.3 Session Bus Vs System Bus:
6.4 D-Bus Applications
6.5 D-Bus Architecture
6.6 D-Bus Use Cases
6.6.1 D-Bus Advantages and Disadvantages
6.6.2 Performance
6.6.3 Comparison with other IPC Mechanisms
6.7 System D-Bus
6.7.1 System D-Bus: The Core Basics
6.7.2 Security Policies (Policy Files) in System D-Bus
6.7.3 Key Functions of System D-Bus
6.7.4 Security and Considerations:
6.7.5 Programmatic Interaction with System D-Bus:
6.7.6 System D-Bus using QT C++:
6.8 . Challenges and Future Directions
Module C
CHAPTER 7:Vision Assistance System
7.1 Introduction
7.2 Problem Statement
7.3 System Objectives

7.4 Working Methodology .....
7.5 System Requirements .....
7.6 Why? .....
7.6.1 Why Use Artificial Intelligence (AI)? .....
7.6.2 Why YOLOv4-Tiny? .....
7.6.3 Why Raspberry Pi and Not STM32? .....
7.7 Challenges .....
7.8 System Extensions and Capabilities .....
7.9 System Block Diagram .....
7.10 System Flowchart .....
CHAPTER 8: LDW (Lane Departure Warning) .....
8.1 Introduction [40] .....
8.2 Problem Statement .....
8.3 System Description .....
8.4 System Requirements .....
8.5 Expected Results .....
8.6 Flowchart .....
CHAPTER9: FOTA (Firmware Over The Air) .....
9.1 Introduction .....
9.1.1 Overview of Firmware Updates: .....
9.1.2 Concept of FOTA (Firmware Over-The-Air): .....
9.1.3 Tools and Platforms Used in the Project: .....
9.1.4 Project Objectives: .....
9.1.5 Thesis Outline: .....
9.1.6 What is FOTA? .....
9.1.7Examples of FOTA .....
9.2 Theoretical Foundations of FOTA .....
9.3 Mender Tool - Details and Technologies .....
9.3.1 Introduction to Mender Tool .....
9.4 FOTA Implementation on Raspberry Pi using Mender .....
9.5 Mender Integration with STM32 .....
9.6 Challenges, Solutions, Results, and Conclusion .....
9.7 Flow chart .....
CHAPTER 10: Logger (DTCHandler) .....

10.1 Introduction to Diagnostic Trouble Codes (DTCs) .....
10.2 DTC Handler Design and Architecture [63] .....
10.3 Using and Testing DTC Handler .....
10.4 Future Expansion and Enhancements[65] .....
10.4.1 Data Persistence .....
10.5 System Flow chart .....
Module D .....
CHAPTER 11: Embedded Linux .....
11.1 Introduction .....
11.2 Linux .....
11.2.1 Linux History .....
11.2.2 Linux History .....
11.2.3 Difference between Linux and Windows .....
11.3 What is Embedded Linux? .....
11.4 Embedded Linux vs. Bare metal vs. RTOS .....
11.5 Embedded Linux Architecture .....
11.5.1 Toolchain .....
11.5.2 Bootloader .....
11.5.2.1 Embedded Linux Boot flow Process .....
11.5.3 Linux Kernel .....
11.5.4 Device Tree .....
11.5.5 Root File systems .....
11.5.6 Applications .....
CHAPTER 12: Embedded Linux Development .....
12.1 Introduction .....
12.2 SD card structure in Embedded Linux .....
12.3 The Traditional Method for building a Customized Linux Image .....
12.3.1 Studying the booting sequence .....
12.3.2 Build or download a Toolchain .....
12.3.3 Building a bootloader (U-boot) .....
12.3.4 Building a Linux Kernel .....
12.3.5 Building the Filesystem .....
12.3.6 Building the Application .....
12.4 Using Yocto to build The Image .....

12.4.1 Poky .....
12.4.2 Board Support Package (BSP) .....
12.4.3 Metadata .....
12.4.4 Open-Embedded Core (OE-Core) .....
12.4.5 BitBake .....
12.4.6 Layer .....
12.4.7 Packages .....
12.4.8 QEMU .....
CHAPTER 13: Main System (GUI) .....
13.1 What is GUI .....
13.2 Why do we need GUI in cars? .....
13.3 What is QML? .....
13.4 System Requirements .....
13.5 System Implementation .....
CHAPTER 14:Embedded Linux Development using Yocto Project .....
14.1 Preparing the build Host system .....
14.2 Downloading the Poky Source Code .....
14.3 Creating active Build directory .....
14.4 Building Our First minimal image .....
14.5 Building Our Raspberry pi image .....
CHAPTER 15: Programming Enhancements .....
15.1 Why Choose C++? .....
15.2 What is Qt? .....
15.3 Why Choose Qt? .....
15.4 Why do we use C++ instead of C language? .....
CHAPTER 16: Future work and conclusion .....
16.1 Future Work .....
16.2 Conclusion .....
References .....
بالعربية الملخص .....
أولا: المسجل نظامً .....
ثانيا: الهواء عبر البرامج تحديثً .....
ثالثا: الاصطناعي الذكاءً .....
رابعا: والمكونات التحتية البنية .....

List of figures

Introduction 0

figure 0.1	graph for communication of subsystems	page 20
figure 0.2	The modules of project	page 22

Chapter 1

figure 1.1	STM32f103c8t6 (Blue Pill)	page 27
figure 1.2	The define for each pin in STM32	page 33

Chapter 2

figure 2.1	STM32f103c8t6 (Blue Pill)	page 37
figure 2.2	The define for each pin in STM32	page 39
figure 2.3	Processor (CPU & GPU) & RAM(Memory)	page 42
figure 2.4	Ethernet controller	page 43
figure 2.5	USB Controller	page 43

figure 2.6	Power circuitry	page 44
figure 2.7	The layout of the GPIO header	page 44

Chapter 3

figure 3.1	The component of IR sensor	page 48
figure 3.2	HW 201 Infrared (IR) sensor module	page 49
figure 3.3	How IR sensor works	page 50
figure 3.4	Buzzer	page 52
figure 3.5	DC motor	page 54
figure 3.6	How DC motor works	page 55
figure 3.7	H-Bridge and its component	page 58
figure 3.8	Infrared night vision camera module	page 61
figure 3.9	Bluetooth module HC-05	page 64

Chapter 4

figure 4.1	Car Dimensions Chart	page 67
figure 4.2	First floor of the car and its component	page 68
figure 4.3	Second floor of the car and its component	page 69
figure 4.4	Third floor of the car	page 70
figure 4.5	The carrier of both wheel and motor	page 71
figure 4.6	Overview of The design of the car	page 71
figure 4.7	Horizontal view of the car	page 72
figure 4.8	Horizontal view of the car	page 72

Chapter 5

figure 5.1	Communication system in our smart car prototype	page 76
figure 5.2	CAN Network message format	page 78
figure 5.3	CAN frame format	page 79
figure 5.4	Data frame	page 80

figure 5.5	Error frame	page 80
figure 5.6	Overload frame	page 81

Chapter 6

figure 6.1	QT D-Bus Communication scheme	page 99

Chapter 7

figure 7.1	Vision system block diagram	page 105
figure 7.2	Vision system flowchart	page 106

Chapter 8

figure 8.1	Flowchart of LDW system	page 109

Chapter 9

figure 9.1	ST-Link Programmer	page 116
figure 9.2	Mender tool	page 118
figure 9.3	Raspberry Pi using Mender	page 120
figure 9.4	Mender Integration with STM32	page 122
figure 9.5	FOTA system flowchart	page 124

Chapter 10

figure 10.1	Logger device	page 126
figure 10.2	Code (Header file - DTC handler.cpp)	page 127
figure 10.3	Code (source file - DTC handler.cpp)	page 120
figure 10.4	Logger system (full system diagnostic)	page 132
figure 10.5	Logger system flowchart	page 132

Chapter 11

figure 11.1	Embedded linux architecture	page 141
figure 11.2	Cross toolchain	page 142
figure 11.3	Cross compilation toolchain	page 143
figure 11.4	Embedded system boot process	page 144
figure 11.5	Layered operating system architecture	page 145
figure 11.6	Root file systems	page 147

Chapter 12

figure 12.1	Yocto project logo	page 154
figure 12.2	Yocto Project Environment	page 156
figure 12.3	Yocto project component and tools	page 156
figure 12.4	Download poky to the Host Machine and its file structure contains	page 159

Chapter 13

figure 131	Main window	page 164

figure 13.2	WIFI window	page 166
figure 13.3	WIFI window-on case	page 167
figure 13.4	Password wifi window	page 169
figure 13.5	Control window	page 170
figure 13.6	Camera window	page 171
figure 13.7	Applications Window	page 172
figure 13.8	Restart window	page 174

Chapter 14

figure 14.1	Contents inside the poky directory	page 176
figure 14.2	Sourcing the build environment	page 177
figure 14.3	Build directory directories	page 177
figure 14.4	Conf content	page 179
figure 14.5	tmp directories	page 180

INTRODUCTION

Road safety becomes a major public health concern when the statistics show that more than 3,000 people around the world succumb to death daily due to road traffic injury. Also, road crashes lead to global economic losses as estimated road traffic injury costs US$518 billion per year. The huge Economic losses are an economic burden for developing countries. It is reflected that the road crash costs are estimated to be US$ 100 billion in developing countries which is twice the annual amount of development aid to such countries. Over 130,000 fatalities and nearly 500,000 people were permanently disabled due to road crashes over past decades. The economic losses due to the road crashes are considerably high, costing approximately US$2,500 million per year (about US$0.3 million per hour), or 3.4% of the Gross National Product (GNP). The collection and use of accurate and comprehensive data related to road accidents is very important to road safety management. The road accident data are necessary not only for statistical analysis in setting priority targets but also for in depth study in identifying the contributory factors to have a better understanding of the chain of events. Having the inconsistencies in the aims of the police and the road safety engineers, the data analysis and its interpretation usually does not result in proper countermeasures. Sometimes a lack of proper knowledge of crash and proper training of the police officers in charge of systematic data collection procedures from a crash scene adds to the diverging nature of the role of the police and the road safety

professionals. These problems have become a burning issue for developing countries addressing road safety without completed crash data due to the negligence of the concerned authorities.

Road accidents are the most unwanted thing to happen to a road user, though they happen quite often. The most unfortunate thing is that we don't learn from our mistakes on the road. Most of the road users are quite well aware of the general rules and safety measures while using roads but it is only the laxity on part of road users, which causes accidents and crashes. The main cause of accidents and crashes are due to human errors. We are elaborating some of the common behavior of humans which results in accidents.

Accidental Lane Drifting and Departures:

Drivers are not infallible. Even the most vigilant can experience momentary lapses In attention, perhaps due to internal distractions (like adjusting the radio, checking navigation screen, or engaging in conversation), external distractions (like billboards or roadside events), or even physiological factors like fatigue and drowsiness. These factors lead to a subtle, often unconscious, drifting of the vehicle outside its intended lane markings. Such deviations are incredibly dangerous because they can result in:

- Head-on collisions: If drifting into an oncoming traffic lane.
- Sideswipe accidents: If drifting into an adjacent lane occupied by another vehicle.
- Run-off-road accidents: If drifting onto the shoulder or off the pavement, which can lead to rollovers, striking obstacles, or difficult recovery maneuvers.
- Delayed reaction time: Even if a driver becomes aware of the drift, the time taken to recognize and correct it can be too long, especially at higher speeds.

Impaired Driving Performance and Increased Risk in Low-Light or Adverse Visibility Conditions: The human eye is remarkable, but its capabilities are severely limited in darkness. Beyond the range of typical headlights, especially on unlit roads,

drivers struggle to perceive hazards that are crucial for safe navigation. **This "invisible zone" can hide:**

- Pedestrians and cyclists: Who may be wearing dark clothing and are hard to spot until very close.
- Animals: Such as deer or other wildlife that can suddenly dart into the road.
- Unforeseen obstacles: Debris, fallen tree branches, or even stalled vehicles that blend into the darkness.
- Reduced depth perception: Making it harder to judge distances and speeds of other objects.
- Eye strain and fatigue: Constantly straining to see in the dark can accelerate driver fatigue, further diminishing alertness and reaction times. These limitations mean that by the time a driver's headlights illuminate a hazard, there might be insufficient time to react safely, leading to emergency braking, swerving, or a collision.

Lack of Objective Data for Accident Reconstruction, Driver Performance Analysis, and System Improvement.

In the aftermath of an accident, human memory is fallible, subjective, and often incomplete due to the traumatic nature of the event.

Eyewitness accounts can be contradictory, and physical evidence might be ambiguous. For fleet managers, individual drivers, or automotive engineers, understanding why an incident occurred was it driver error, mechanical failure, or an external factor?

0.1 How does our project solve these problems?

we are looking to develop more efficient solutions using modern technology:

1. **FOTA (Firmware Over-the-Air):** Continuous Improvement & Security FOTA enables wireless updates to all system components, ensuring continuous improvement, bug fixes, and enhanced performance for all safety features throughout the vehicle's lifespan. This keeps the system effective and secure without user inconvenience.
2. **Logger: Data-Driven Safety & Analysis** The Logger continuously records detailed vehicle operational data (speed, braking, steering) to provide objective

"black box" information. This data is used for accurate accident reconstruction, identifying risky driver behaviors, and validating system performance to inform training and policy improvements

3. **ADAS (Advanced Driver-Assistance Systems): LDW & Vision -Proactive Accident Prevention:**
    a. Lane Departure Warning (LDW): Uses cameras and algorithms to detect unintentional lane departures and provides immediate, multimodal alerts to re-engage the driver, preventing collisions and run-offs.
    b. Night Vision Assist: Employs thermal/infrared cameras to extend visibility beyond headlights, highlighting hidden hazards (pedestrians, animals, obstacles) at greater distances, providing crucial extra reaction time.

0.2 Project Descriptions

- The project represents an innovative leap in the integration of real-time diagnostics, firmware management.
- Intelligent driver-assistance technologies in modern automotive systems.
- As vehicles increasingly rely on embedded electronics and software-driven functions, the necessity for robust monitoring and autonomous system optimization becomes more urgent.
- This project addresses these challenges by developing a smart vehicle architecture empowered by a Logger system, Firmware Over-The-Air (FOTA) updates, and AI-driven ADAS (Advanced Driver Assistance Systems) including Lane Departure Warning (LDW) and Night Vision systems.

0.3 Project Objectives

- **Design and Develop a Real-Time Diagnostic Logger:**
    - Continuously monitor ADAS subsystems like LDW and Vision for faults or anomalies.
    - Log and communicate error codes and failure details to the main system
- **Implement Secure FOTA Mechanism:**

- Create a secure pipeline for wirelessly updating vehicle firmware using ESP8266 modules.
- Ensure update authenticity via cryptographic validation and rollback

- **Develop AI-Based Vision and Lane Departure Warning Systems:**
  - Integrate thermal and infrared camera inputs with AI processing using OpenCV and TensorFlow on Raspberry Pi.
  - Enable real-time object and lane detection with feedback systems to assist the driver.
- **Integrate the Logger, AI, and FOTA Systems into a Unified Control Architecture:**
  - Manage subsystem operations using STM32 microcontrollers.
  - Coordinate inter-subsystem communication using the CAN protocol.
- **Create a Custom Diagnostic Protocol (UDS-lite):**
  - Implement essential UDS services like reading/clearing DTCs and data identification using Service 0x19, 0x22, and 0x31.

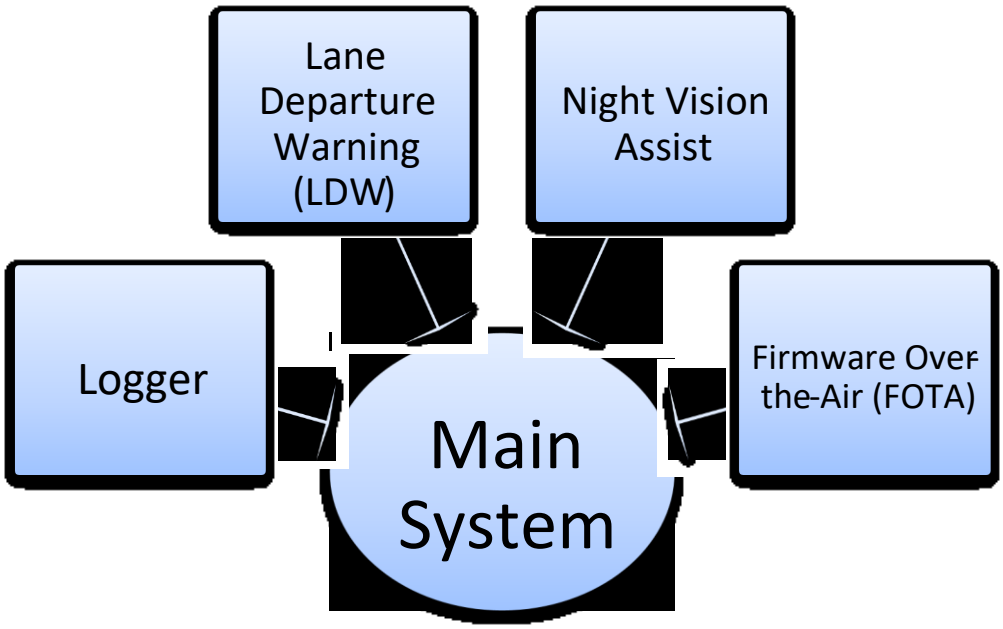


Figure 0.1

0.4 Expected Outcomes

1.Improved Vehicle Safety and Reliability:

a. The system will proactively detect and report issues, preventing accidents caused by undetected subsystem failures.

b. Real-time alerts and updates ensure that the driver is always informed about the vehicle's condition.

2. Efficient Maintenance Through Remote Updates:

a. The FOTA system allows manufacturers and service centers to update vehicle firmware remotely, minimizing the need for manual servicing.

3. Enhanced Driver Assistance via AI:

a. AI-powered Vision and LDW systems reduce risks associated with nighttime driving and unintentional lane departures.

4. Lower Operational Costs for Fleet Managers:

a. Centralized data logging and cloud diagnostics enable predictive maintenance and efficient fleet management.

5. Educational and Research Contribution:

a. The project serves as a scalable platform for future work in autonomous vehicle development and embedded systems research.

0.5 Research Methodology Implementation and Validation:

The methods that will use to finish this project are enlisted here:

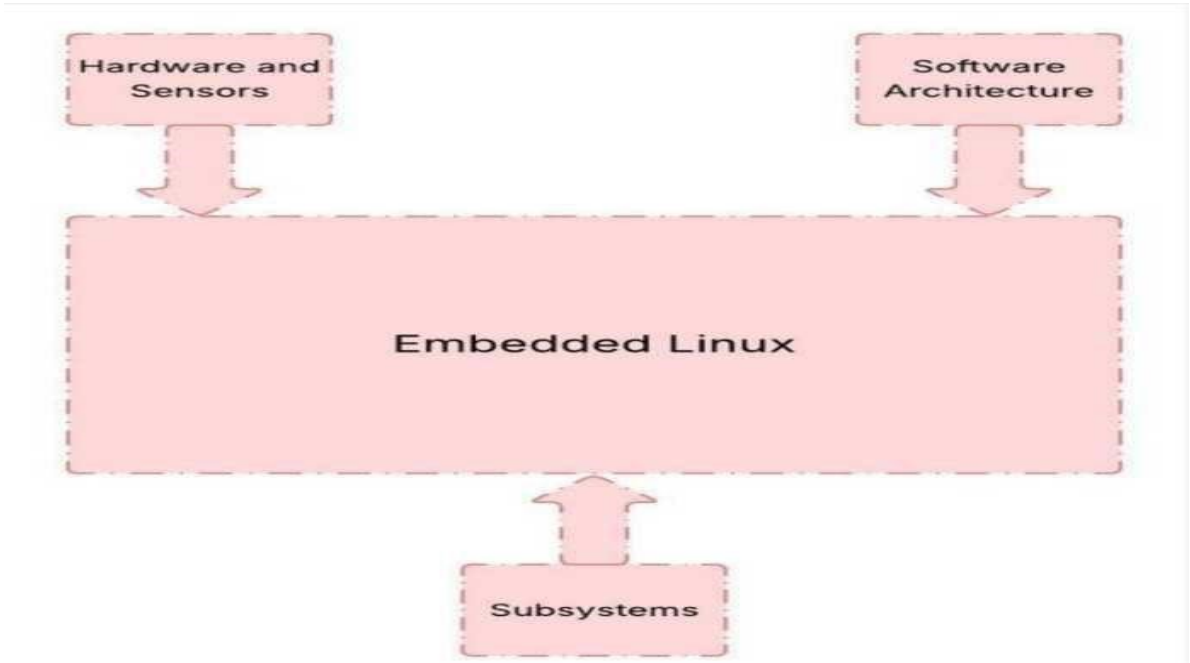
- Learn the basic principle of interfacing and coding the drivers, then implement it in the project. Learn the basic principle of Automotive and How a car works, then implement it in the project to control all cases of road.
- Learn the basic principle of ARM and interfacing with blue pill, then implement it in the project.
- Learn the basic principle of Linux administration and interfacing with RPI, then implement it in the project

- Programming of microcontrollers to control the I/O devices. Circuits developments and design for the robot control unit.
- Making hardware prototypes and testing the robot function.

0.6 Project Architecture

Our Project contains 4 modules:

1. **Hardware and Sensors**
2. **Software Architecture**
3. **Subsystems**
4. **Embedded Linux**



We will take about them individually

Figure 0.2

Module A: Hardware and Sensors

We'll explain the hardware used, the ECU used and components of it, the communication of ECU and the components, the Sensors which are connected to the overall system.

This module consists of three chapters:

Chapter 1: STM32F103C8T6.

Chapter 2: The Raspberry Pi.

Chapter 3: The sensors & actuators

Chapter 4: The Design

Module B: Software Architecture

We'll explain the software architecture used, which is the communication between the two blue pills and the two-raspberry pi.

-This module consists of three chapters:

Chapter 5: Controller Area Network (CAN)

Chapter 6: DBus system

Module C: Subsystems

We'll explain the Subsystems we used to solve the problem of collision and crash, these are two applications (LDW, Vision, Logger ,Dbus and FOTA) and explain the function of each subsystem and the role of it in the system overall. -This module consists of four chapters:

Chapter 7: LDW System

Chapter 8: Vision System

Chapter 9: Logger System

Chapter 10: FOTA System

Module D: Embedded Linux

Chapter 11: Embedded Linux

Chapter 12: Embedded Linux Development

Chapter 13: Embedded Linux Development using Yocto Project

Module A

Hardware & Sensors

Chapter 1 : STM32F103C8T6.

Chapter 2 : Raspberry Pi.

Chapter 3 : Sensors & Actuators.

CHAPTER 1:
BLUE PILL(STM32F103C8T6)

In this chapter, we'll talk about the STM32F103C8T6.Also, we'll discuss why STM32F103C8T6, its Versions, Hardware Component, Interfacing with GPIO pins and Operating System.

1.1 Introduction

The STM32F103C8T6, often referred to simply as the STM32F103 or more commonly as the "Blue Pill" in hobbyist circles, is a popular microcontroller from
STMicroelectronics. It belongs to the STM32F1 series and features an ARM Cortex-M3 core, which is a 32-bit RISC processor capable of executing instructions with high efficiency. This microcontroller is widely used in various applications ranging from industrial automation to consumer electronics and hobbyist projects due to its versatility, performance, and cost-effectiveness.

This board offers a range of features including various sensors, communication interfaces (such as UART, SPI, I2C, CAN), GPIO pins. Its ARM Cortex-M3 processor running at up to 72 MHz, coupled with generous Flash and RAM resources, makes it suitable for a wide range of applications including industrial automation, IoT devices, robotics, and more.

The STM32F103C8T6 is known for its ease of use in both professional and hobbyist environments, thanks to its rich feature set and availability of development resources.

Its affordability and broad compatibility with various software tools and libraries make it a preferred choice for embedded systems development across different industries.

1.2 Why STM32f103c8t6 ?

The STM32F103C8T6 microcontroller is favored by developers and engineers for a variety of reasons, making it a popular choice in the embedded systems industry and among hobbyists alike:

1. **Powerful ARM Cortex-M3 Core:**
    a. The STM32F103C8T6 is based on the ARM Cortex-M3 core, which provides significant processing power (up to 72 MHz) for real-time processing tasks. This makes it suitable for applications requiring efficient computation and fast response times.
2. **Rich Peripherals and Connectivity:**
    a. It integrates a wide range of peripherals including UARTs, SPI, I2C, ADCs, DACs, timers, and more. This versatility allows developers to interface with various sensors, actuators, displays, and communication modules easily.
3. **Memory and Storage Options:**
    a. With 64 KB of Flash memory and 20 KB of SRAM, the STM32F103C8T6 offers sufficient memory for storing program code and data, enabling more complex applications and algorithms.
4. **Low Power Consumption:**
    a. It is designed to operate efficiently in low-power modes, making suitable for battery-powered applications where power consumption is critical.
5. **Cost-Effective Solution:**
    a. The STM32F103C8T6 is cost-effective compared to higher-end STM32 microcontrollers while still offering robust features and performance. This makes it accessible for both commercial and hobbyist projects.

6. Extensive Development Ecosystem:

a. Supported by a comprehensive development ecosystem including STM32CubeMX for graphical MCU configuration and initialization, STM32CubeIDE for integrated development environment, and STM32CubeHAL (Hardware Abstraction Layer) for low-level software development. This ecosystem streamlines the development process and reduces time-to-market.

7. Community Support and Documentation:

a. Due to its popularity, the STM32F103C8T6 has extensive community support, forums, tutorials, and example projects available online. This makes it easier for developers to troubleshoot issues, learn, and share knowledge.

8. Versatility Across Applications:

a. Its versatility makes it suitable for a wide range of applications including industrial automation, consumer electronics, IoT devices, motor control, robotics, and more. Developers can adapt it to various project requirements with relative ease.

In conclusion, the STM32F103C8T6 microcontroller is chosen for its powerful ARM Cortex-M3 core, rich peripherals, low power consumption, cost effectiveness, extensive development tools, and broad application suitability.

These factors collectively make it a preferred choice for embedded systems development, from simple control tasks to more complex and demanding applications.

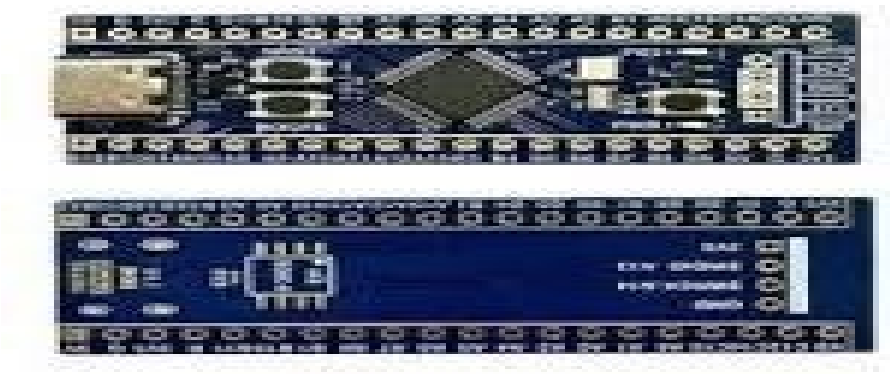


Figure 1.1

1.3 STM32F103C8T6 Board Features

Includes ST state-of-the-art patented technology

- ☐ Arm® 32-bit Cortex®-M3 CPU core

72 MHz maximum frequency, 1.25 DMIPS/MHz (Dhrystone 2.1) performance at 0 wait state memory access to Single-cycle multiplication and hardware division.

- ☐ Memories
  - o 64 or 128 Kbytes of Flash memory o 20 Kbytes of SRAM.
- ☐ Clock, reset and supply management
  - o 2.0 to 3.6 V application supply and I/Os o POR, PDR, and programmable voltage detector (PVD)
  - o 4 to 16 MHz crystal oscillator o Internal 8 MHz factory-trimmed RC o Internal 40 kHz RC - PLL for CPU clock o 32 kHz oscillator for RTC with calibration.
- ☐ Low-power
  - o Sleep, Stop and Standby modes o VBAT supply for RTC and backup registers.
- ☐ 2x 12-bit, 1 ps A/D converters (up to 16 channels)
  - o Conversion range: 0 to 3.6 V 33 o Dual-sample and hold capability
  - o Temperature sensor.

☐ DMA

- o 7-channel DMA controller o Peripherals supported: timers, ADC, SPIs, I 2Cs and USARTs.

☐ Up to 80 fast I/O ports

- o 26/37/51/80 I/Os, all mappable on 16 external interrupt vectors o almost all 5 V-tolerant

- **Debug mode:**
  - ○Serial wire debug (SWD) and JTAG interfaces.
- **Seven timers:**
  - ○ Three 16-bit timers, each with up to 4 IC/OC/PWM or pulse counter and quadrature (incremental) encoder input.
  - ○ 16-bit, motor control PWM timer with dead-time generation and emergency stop.
  - ○ Two watchdog timers (independent and window).
  - ○ SysTick timer 24-bit down counter.
- **Up to nine communication interfaces:**
  - ○ Up to two I2C interfaces (SMBus/PMBus®)
  - ○ Up to three USARTs (ISO 7816 interface, LIN, IrDA capability, modem ontrol)
  - ○ Up to two SPIs (18 Mbit/s)
  - ○ CAN interface (2.0B Active)
  - ○ USB 2.0 full-speed interface
- **CRC calculation unit, 96-bit unique ID**
- **Packages are ECOPACK®**

1.4 STM32F103C8T6 Board Hardware

In this Part, we will talk about the STM32F103C8T6 Specifications Pin Diagram and Description which we used in our project. Also, we will discuss the Layout and Interfacing of GPIO pins and Hardware Components of it.

1.4.1 Hardware component of STM32F103C8T6

STM32F103C8T6 provides the following hardware components:

1.ARM Cortex-M3 Core:

The heart of the STM32F103C8T6 is its ARM Cortex-M3 processor core, which operates at speeds up to 72 MHz. This 32-bit core provides efficient processing power suitable for real-time applications.

2.Memory:

a.Flash Memory: The STM32F103C8T6 typically includes 64 KB of Flash memory for storing program code. This non-volatile memory retains its contents even when the power is turned off.

b.SRAM (Static Random Access Memory): It features 20 KB of SRAM for temporary data storage during program execution. SRAM provides faster access times compared to Flash memory.

3.Clock and Reset Management:

a.Internal Oscillators: The microcontroller includes internal RC oscillators and may support external crystal oscillators for accurate timing and clock generation.

b.Reset Circuitry: Includes hardware reset sources such as power-on reset (POR), brownout reset (BOR), and external reset pins for system initialization and recovery.

4.Peripherals:

a.The STM32F103C8T6 integrates a variety of peripherals, including but not limited to

- **Timers:** General-purpose timers, advanced-control timers (PWM), and watchdog timers.
- **Communication Interfaces:** UARTs (Universal Asynchronous Receiver/Transmitter), SPI (Serial Peripheral Interface), I2C (InterIntegrated Circuit), CAN (Controller Area Network), and USB (Universal Serial Bus).

- **Analog-to-Digital Converters (ADCs):** For converting analog signals from sensors or other sources into digital data.
- **Digital-to-Analog Converters (DACs):** Some variants may include DACs for generating analog signals.
- **GPIO (General-Purpose Input/Output):** For interfacing with external devices and peripherals.

1. **Power Management**:
   a. Includes power supply circuitry for various voltage levels (typically 3.3 V and sometimes 5 V tolerant I/O), and low-power modes to optimize energy consumption.
2. **Debug and Programming Interfaces**:
   a. JTAG/SWD (Serial Wire Debug): Used for programming and debugging the microcontroller.
   b. Bootloader Support: Allows firmware updates and programming via different interfaces (USB, UART, etc.) depending on the specific board configuration.
3. **Package and Pin Configuration:**
   a. The STM32F103C8T6 is available in various package options (such as LQFP48) with a defined number of I/O pins for connecting to external components and circuits.

1.4.2 Layout the GPIO Headers

The GPIO (General-Purpose Input/Output) headers on the STM32F103C8T6, commonly found on development boards like the "Blue Pill," typically have a specific layout. Here's a general description of how these GPIO headers is organized:

1. **Pin Headers:**
   a. The STM32F103C8T6 development board usually has two rows of pin headers, one on each side of the board.
   b. Each header row typically consists of 20 pins (10 pins per row).
2. **Pin Configuration:**

a. **Top Row (Typically Even Pins):** Pins are numbered consecutively from 2 to 40 (for a total of 20 pins).
b. **Bottom Row (Typically Odd Pins):** Pins are also numbered consecutively, usually from 1 to 39 (for a total of 20 pins).

3. Pin Functions:

a. The pins on these headers serve various functions, including GPIO, power supply, ground, and communication interfaces (such as UART, SPI, I2C).
b. **GPIO Pins**: Several of these pins are dedicated to GPIO functions, allowing digital input and output operations.

4. Power Supply and Ground Pins:

a. Typically, pins 1 and 2 (bottom row) are connected to VCC (3.3 V) and GND respectively.
b. Other pins might also be designated for specific power supply voltages (e.g., 5 V, VIN) and additional ground connections.

5. Communication Interfaces:

a. Some pins might be multiplexed to support serial communication interfaces like UART, SPI, and I2C.
b. These pins often have dual functionalities, where they can be configured either as GPIO or for specific communication protocols.

6. Labeling and Documentation:

a. The pin headers are usually labeled on the PCB itself or in the accompanying documentation.
b. The STM32 microcontroller datasheet and reference manual provide detailed pinout diagrams and descriptions, which are crucial for understanding the exact functions and capabilities of each pin.

7. Connection to External Components:

Users can connect external sensors, actuators, displays, and other peripherals directly to these GPIO headers, enabling easy prototyping and development of embedded systems.

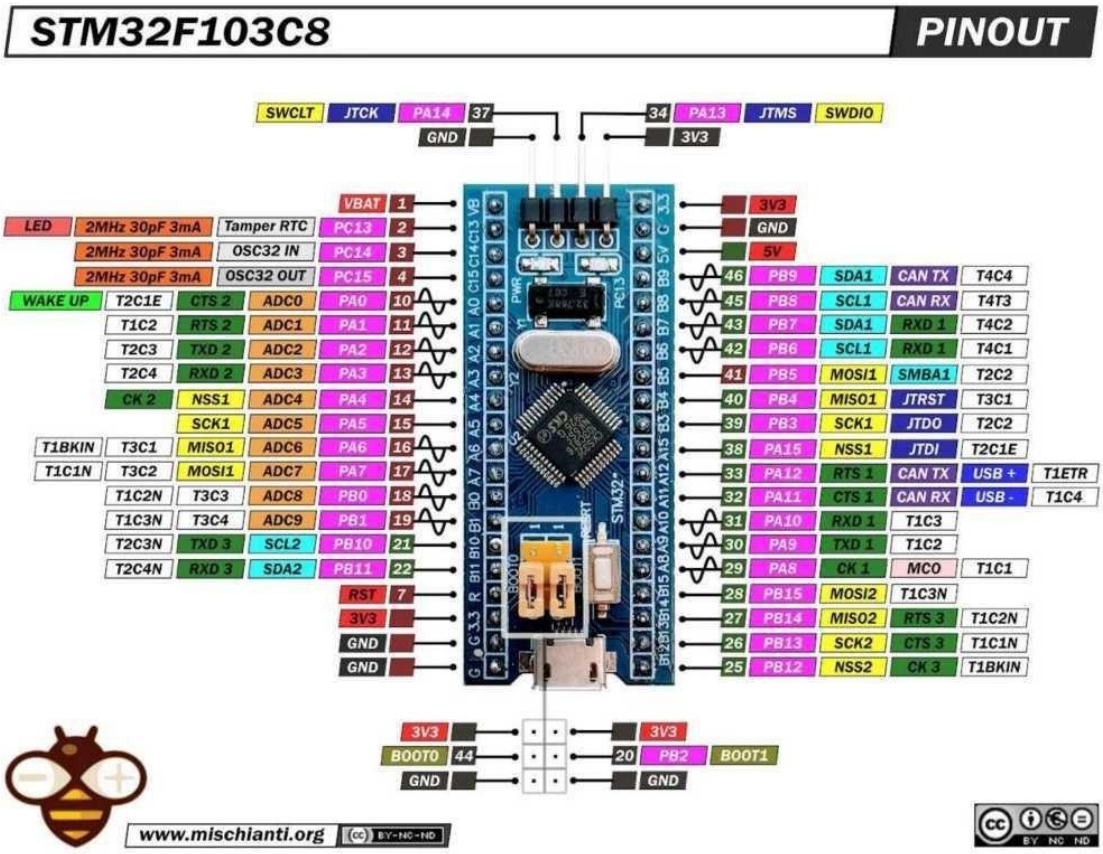


Figure 1.2

1.5 STM32F103C8T6 OS

The STM32F103C8T6 microcontroller itself does not have a built-in operating system (OS) in the traditional sense like desktop or server operating systems such as Windows or Linux. Instead, it is typically used in embedded systems where it runs a real-time operating system (RTOS) or operates in a bare-metal (no OS) environment, depending on the application requirements.

Here are the common scenarios for using an STM32F103C8T6 microcontroller with different types of software environments:

1. Bare-Metal Programming

a. Many applications for the STM32F103C8T6 involve writing firmware directly for the microcontroller without an operating system. In bare-metal

programming, developers directly control hardware peripherals and manage tasks and timing within their application code.

2. Real-Time Operating Systems (RTOS):

a. For more complex applications requiring multitasking, task scheduling, and better resource management, developers may opt to use an RTOS. Popular RTOS choices for STM32 microcontrollers include FreeRTOS, RTX (from Keil), and STM32CubeRTOS. These RTOS provide scheduling mechanisms, synchronization primitives, and other features to simplify development of complex embedded systems.

3. Embedded Frameworks and Libraries:

a. STM32 microcontrollers are often programmed using vendor provided frameworks and libraries such as STM32CubeMX and STM32 HAL (Hardware Abstraction Layer). These frameworks simplify configuration and use of microcontroller peripherals like timers, UARTs, ADCs, etc., abstracting low-level hardware details.

In summary, while the STM32F103C8T6 microcontroller itself does not come with a full-fledged operating system, it is highly versatile and supports various software environments ranging from bare-metal programming to real-time operating systems. The choice of software environment depends on factors such as application complexity, real-time requirements, resource constraints, and developer preference.

CHAPTER 2 : RASPBERRYPI

In this chapter we will discuss what is the raspberry pi, why Raspberry pi, its Versions, Hardware Component, Interfacing with GPIO pins and Raspberry pi Operating System.

2.1 Introduction

The Raspberry Pi is probably the most popular electronics prototyping platform ever. Its creators, the Raspberry Pi Foundation, wanted to create a single-board computer that could be used to teach computer science and programming to high school students in developing countries.

2.1.1 What is a Raspberry Pi?

Raspberry Pi is a low-cost, credit card-sized computer that was first developed in 2012 by the Raspberry Pi Foundation in the UK.

It can connect to a huge variety of sensors and other modules like LCD displays, servos, and motors.

The Raspberry Pi was designed to be a tool to promote the teaching of basic computer science in schools and developing countries.

The original Raspberry Pi was equipped with a 700 MHz ARM11 processor, 256 MB of RAM, and had a single USB port, however, today's models are much more powerful, with quad-core processors, up to 8GB of RAM, and multiple USB and HDMI ports.

The Raspberry Pi can run a variety of operating systems, including Linux distributions, Windows 10, and even Android, it can be used for a wide range of projects, from simple tasks such as browsing the web and playing games, to more complex projects such as controlling robots, home automation, and even building a supercomputer. Overall, the Raspberry Pi has become an incredibly versatile and useful tool for

anyone interested in learning about computing and electronics. There Are multiple types of mobile robotics such as wheeled, tracked, legged, flying, and underwater robots.

We used the Raspberry Pi 4 Model B in our project:

The Raspberry Pi 4 represents a major upgrade over previous versions. It provides true desktopclass performance in a compact, affordable, and energy-efficient board. Whether you're editing documents, browsing the web with many tabs open, managing spreadsheets, or building presentations, the Raspberry Pi 4 delivers a smooth and responsive experience making it an excellent choice for embedded systems and development projects.

Raspberry Pi 4 Features can list as:

1. **Silent and Energy-efficient:** Like earlier models, Raspberry Pi 4 operates without a fan (though optional cooling is recommended under high load), consuming significantly less power than traditional PCs — ideal for 24/7 embedded applications
2. **Fast Networking:** It comes with true Gigabit Ethernet (not over USB as in previous models), offering high-speed network connectivity. It also features dual-band 802.11ac WiFi and Bluetooth 5.0, making it ideal for IoT and communication-based systems.
3. **USB 3.0 Support:** Raspberry Pi 4 includes two USB 3.0 ports and two USB 2.0 ports, allowing up to 10x faster data transfer speeds compared to USB 2.0 — useful for connecting fast storage devices, cameras, and peripherals.
4. **Multiple RAM Options:** Raspberry Pi 4 is available in 2GB, 4GB, and 8GB LPDDR4 RAM variants, allowing users to select the right configuration depending on their memory requirements.

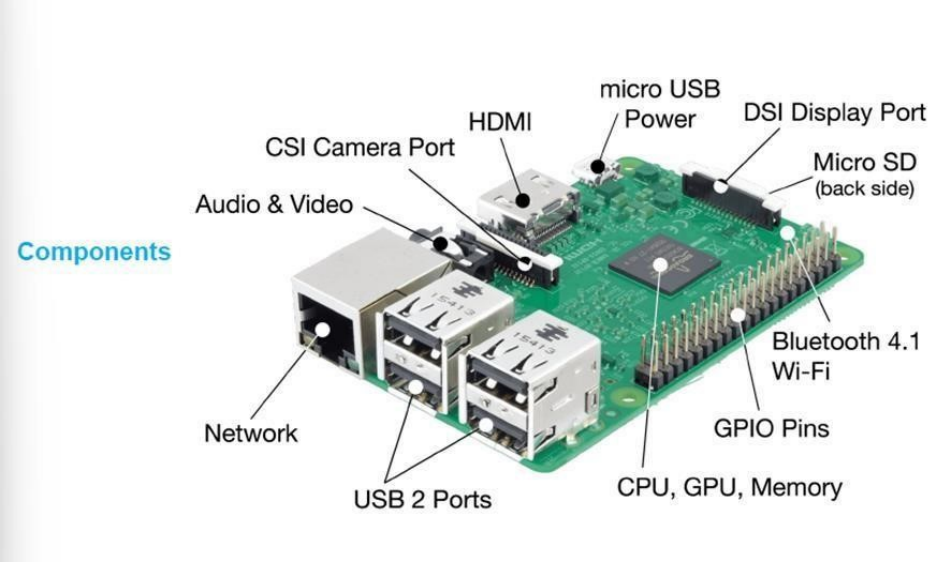


Figure 2.1

2.2 Why Raspberry Pi?

Raspberry Pi has many advantages that make it a popular choice for a wide range of projects and applications. Here are some reasons why Raspberry Pi is a great choice:

- Low cost (Cheap): Raspberry Pi is a low-cost computer, making it an affordable option for beginners or for use in projects where cost is a concern.
- Small Size: The Raspberry Pi is small and lightweight, so it's easy to carry around and fit into tight spaces.
- Low Power Consumption: Raspberry Pi uses very little power, making it an energy-efficient option.
- Flexibility: The Raspberry Pi can run a variety of operating systems and applications, making it a versatile tool for a wide range of projects.
- GPIO Pins (easy access to the GPIO): The Raspberry Pi's GPIO pins make it easy to connect to a wide range of sensors and devices, allowing you to build custom electronics projects.
- Community Support: Raspberry Pi has a large and active community of users and developers who share knowledge and resources, making it easy to get help and find solutions to problems.

- Educational Value: Raspberry Pi was designed as a tool to promote the teaching of basic computer science in schools and developing countries, so it's a great way to learn about computing and electronics.
- Complete computing platform.
- Runs Linux (MMU).

Overall, Raspberry Pi is a cost-effective, flexible, and versatile tool that can be used for a wide range of projects and applications, making it a popular choice for hobbyists, students, and professionals alike.

2.3 Why Raspberry Pi Versions?

Here are the different versions of Raspberry Pi that have been released so far, each version of the Raspberry Pi has its own unique features and capabilities, and they can be used for a wide range of applications, including education, hobby projects, and industrial automation:

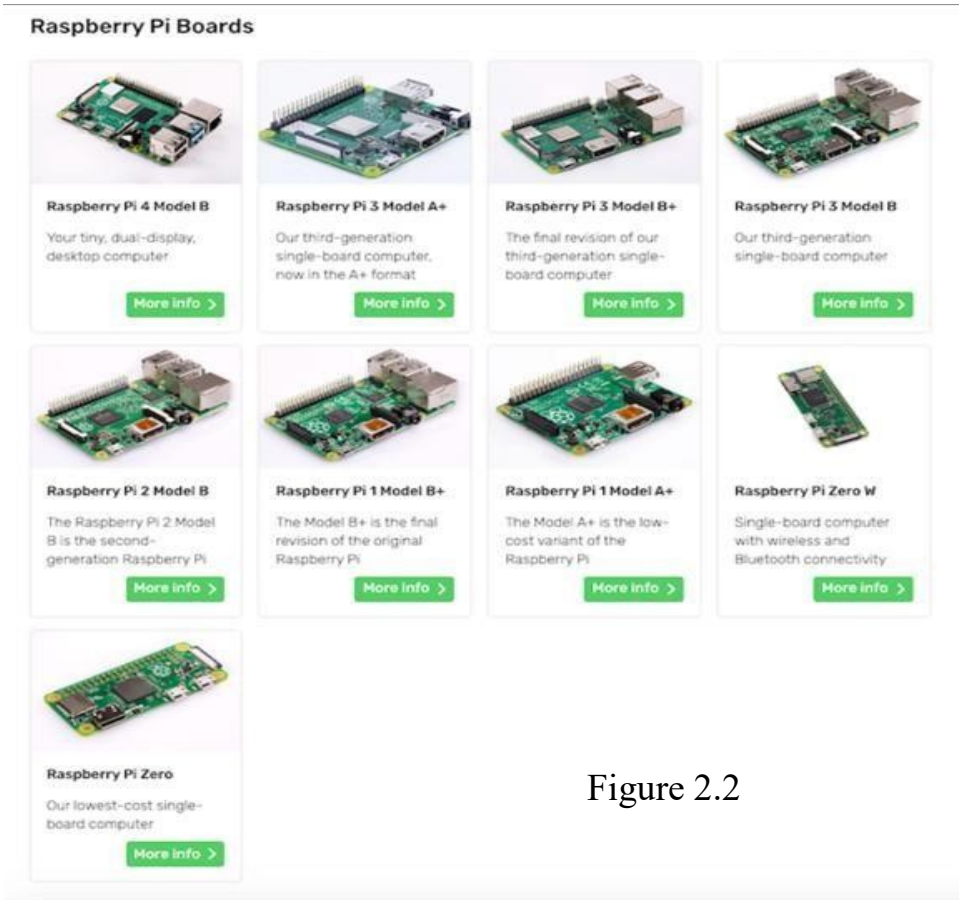

Raspberry Pi Boards
Raspberry Pi 4 Model B
Your tiny, dual-display, desktop computer
More info
Raspberry Pi 3 Model A+
Our third-generation single-board computer, now in the A+ format
More info
Raspberry Pi 3 Model B+
The final revision of our third-generation single-board computer
More info
Raspberry Pi 3 Model B
Our third-generation single-board computer
More info
Raspberry Pi 2 Model B
The Raspberry Pi 2 Model B is the second-generation Raspberry Pi
More info
Raspberry Pi 1 Model B+
The Model B+ is the final revision of the original Raspberry Pi
More info
Raspberry Pi 1 Model A+
The Model A+ is the low-cost variant of the Raspberry Pi
More info
Raspberry Pi Zero W
Single-board computer with wireless and Bluetooth connectivity
More info
Raspberry Pi Zero
Our lowest-cost single-board computer
More info

Figure 2.2

- Raspberry Pi Model A: Released in 2012, it was the first Raspberry Pi board and featured a 700MHz ARM11 processor, 256MB of RAM, one USB port, and no Ethernet port.
- Raspberry Pi Model B: Released in 2012, it was a higher-end version of the Model A and featured a 700MHz ARM11 processor, 512MB of RAM, two USB ports, and an Ethernet port.
- Raspberry Pi Model A+: Released in 2014, it was a smaller and cheaper version of the Model A, with 256MB of RAM, one USB port, and no Ethernet port.
- Raspberry Pi Model B+: Released in 2014, it featured a 900MHz quad-core ARM CortexA7 processor, 1GB of RAM, four USB ports, and an Ethernet port.
- Raspberry Pi 2 Model B: Released in 2015, it featured a 900MHz quad core ARM CortexA7 processor, 1GB of RAM, four USB ports, and an Ethernet port.

- Raspberry Pi Zero: Released in 2015, it was a smaller and cheaper version of the Model A, with a single-core 1GHz ARM11 processor, 512MB of RAM, and a micro-USB port.
- Raspberry Pi 3 Model B: Released in 2016, it featured a 1.2GHz 64-bit quad-core ARM Cortex-A53 processor, 1GB of RAM, four USB ports, and an Ethernet port.
- Raspberry Pi Zero W: Released in 2017, it was the same as the Zero but with added wireless connectivity, including Wi-Fi and Bluetooth.
- Raspberry Pi 3 Model B+: Released in 2018, it featured a 1.4GHz 64-bit quad-core ARM Cortex-A53 processor, 1GB of RAM, four USB ports, and an Ethernet port.
- Raspberry Pi 4 Model B: Released in 2019, it featured a 1.5GHz 64-bit quad-core ARM Cortex-A72 processor, up to 8GB of RAM, two USB 2.0 ports, two USB 3.0 ports, and two micro-HDMI ports.
- Raspberry Pi Compute Module 3: Released in 2017, it was designed for industrial applications and included a 1.2GHz quad-core ARM Cortex-A53 processor, 1GB of RAM, and 4GB of eMMC flash storage.
- Raspberry Pi 400: Released in 2020, it featured a Raspberry Pi 3 board built into a compact keyboard, with 4GB of RAM, two USB 3.0 ports, and one USB 2.0 port.
- Raspberry Pi Compute Module 4: Released in 2020, it was the latest version of the Compute Module and included a choice of processors, up to 8GB of RAM, and various storage options.

2.4 Raspberry Pi Hardware[7]

In this Part, we'll talk about the Raspberry pi 4 Specifications Pin Diagram and Description which we used in our project.

Also, we'll discuss the Layout and Interfacing of GPIO pins and Hardware Components of Raspberry pi:

2.4.1 Specifications Pin Diagram and Description

The Raspberry Pi 4 Model B is a powerful and versatile single-board computer ideal for applications such as robotics, automation, image processing, and AI. Below are its updated specifications and pin configuration details:

- Processor: Broadcom BCM2711, 64-bit quad-core ARM Cortex-A72 (ARM v8-A), clocked at 1.5GHz
- GPU: Broadcom VideoCore VI (VC6), capable of:
  - Dual 4Kp30 HDMI display output (via 2× micro-HDMI ports)
  - H.265 (HEVC) decoding at 4Kp60
  - H.264 decoding and encoding
- RAM Options: Available in 2GB, 4GB, and 8GB LPDDR4-3200 SDRAM
- Networking:
  - Gigabit Ethernet
  - Dual-band Wi-Fi 802.11 b/g/n/ac (2.4GHz and 5GHz)
  - Bluetoo5.0, BLE
- USB Ports:
  - 2 × USB 3.0 ports
  - 2 × USB 2.0 ports
- Video & Sound:
  - 2 × micro-HDMI ports (up to 4kp60 supported on a single port)
  - MIPI DSI display port
  - MIPI CSI camera port
  - 4-pole stereo audio and composite video port
- Storage:
  - microSD card slot for OS and data storage
  - USB boot supported (from Raspberry Pi OS update)
- Power Supply:
  - USB-C power input, 5V/3A
  - Power over Ethernet (PoE) enabled using PoE HAT
- GPIO:

- 40-pin GPIO header, fully backward compatible with previous models

- Other features:
  - Full-size HDMI replaced with 2× micro-HDMI ports
  - Improved thermals and power efficiency

2.4.2 Hardware Components of Raspberry Pi

The Raspberry Pi 4 is a credit card-sized single-board computer used in a wide variety of applications, including robotics, home automation, media centers, and IoT systems. It includes several integrated hardware components that work together to provide powerful computing capabilities in a compact and affordable package.
Below is a breakdown of its key hardware components:

1. Processor (CPU & GPU):

   The Raspberry Pi 4 features a Broadcom BCM2711 SoC that includes:

   - CPU: Quad-core ARM Cortex-A72 (ARM v8-A, 64-bit) running at 1.5GHz, offering significant performance improvements over previous versions.
   - GPU: Broadcom VideoCore VI running at 500MHz, capable of:
     - Dual 4Kp30 HDMI output (via two micro-HDMI ports)
     - H.265 (HEVC) decoding at 4Kp60 and H.264 decoding/encoding.

Figure 2.3

2. RAM(Memory):
   - Available in 2GB, 4GB, and 8GB versions
   - Uses LPDDR4-3200 SDRAM
   - Offers higher bandwidth and energy efficiency than older DDR2/DDR3 models
   - Heat sinks or fans are recommended for heavy workloads as the RAM chip can get hot.

3. Ethernet Controller:

- Chipset: Broadcom BCM54213 Gigabit Ethernet controller
- Supports Gigabit Ethernet via a dedicated PCIe lane (not shared over USB like in Pi 3)
- Automatically scales to 100Mbps or 10Mbps if the network speed is lower
- Can be used for:
  - Creating a software-driven router
  - Acting as a media or file server
  - Serving as a bridge between Wi-Fi and Ethernet networks.

Figure 2.4

4. USB Controller

- Chip: VL805 USB 3.0 Host Controller ● Provides:
  - 2 × USB 3.0 ports (up to 5 Gbps)
  - 2 × USB 2.0 ports
- Supports multiple speeds:
  - SuperSpeed (5 Gbps)
  - High-Speed (480 Mbps)
  - Full-Speed (12 Mbps)
  - Low-Speed (1.5 Mbps)
- Capable of powering and connecting multiple USB peripherals simultaneously

Figure 2.5

5. Power Circuitry:

- Raspberry Pi 4 uses a custom Power Management IC (PMIC) system. Earlier models

  like Pi 3 used MaxLinear MXL7704, but Pi 4's power is handled via:

  - USB-C connector supporting 5V/3A
  - Power Over Ethernet (PoE) capability using the PoE HAT
- Efficient power delivery ensures stable operation of all onboard components and peripherals
- Enhanced thermal and voltage management has reduced board complexity and improved performance.

Figure 2.6

2.4.3 Layout of The GPIO Header

The GPIO header allows users to connect a wide range of devices and peripherals to the Raspberry Pi, making it highly versatile and flexible for a wide range of projects.Here is the GPIO Pin Diagram of the Raspberry Pi 4 Model B, showing the 40-pin header layout:

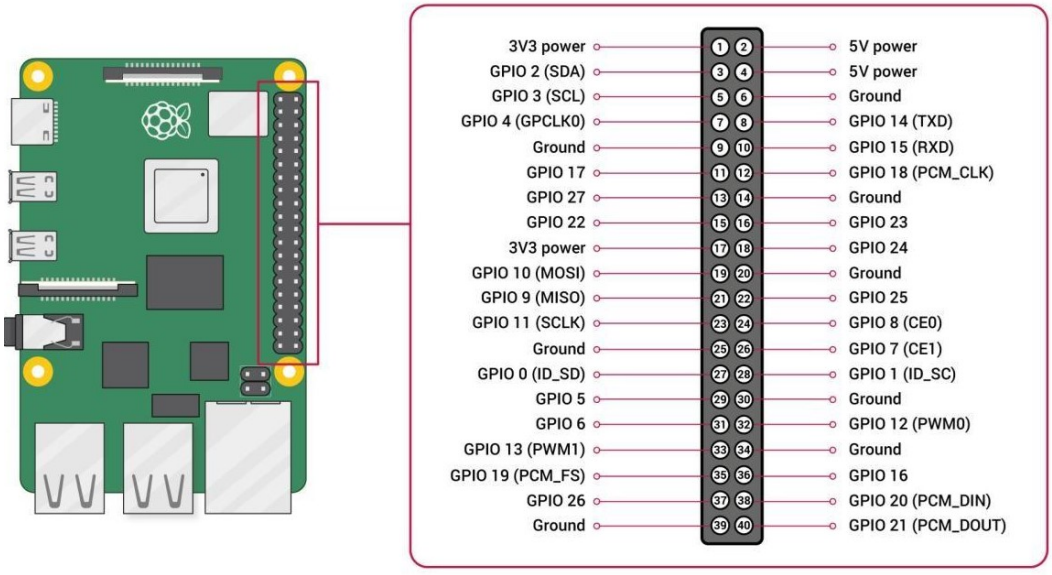


Figure 2.7

2.5 Raspberry Pi OS

Raspberry Pi OS (formerly known as Raspbian) is the official operating system for the Raspberry Pi.

It is a free and open-source operating system based on the Debian Linux distribution.

Raspberry Pi OS is specifically designed for the Raspberry Pi hardware and comes pre-installed with a variety of software.

Here are some of the features and components of Raspberry Pi OS:

1. Desktop Environment: Raspberry Pi OS comes with a desktop environment based on the LXDE desktop environment, which is lightweight and optimized for the Raspberry Pi's hardware.
2. Software: Raspberry Pi OS comes with a variety of pre-installed software, including the Chromium web browser, the Python programming language, and the Thonny Python IDE.
3. Configuration Tools: Raspberry Pi OS includes a variety of tools for configuring and customizing the operating system, including the Raspberry Pi Configuration tool and the raspi-config tool.
4. Command-Line Interface: Raspberry Pi OS includes a powerful command-line interface, which can be used to perform system administration tasks and run scripts.
5. Package Manager: Raspberry Pi OS includes the apt package manager, which can be used to install new software packages and keep the operating system up to date.
6. GPIO Support: Raspberry Pi OS includes support for the Raspberry Pi's GPIO pins, which can be used to interface with external hardware and sensors.

Overall, Raspberry Pi OS is a powerful and flexible operating system that is optimized for the Raspberry Pi hardware. It is a great choice for a variety of projects, from building a media center to creating a home automation system.

2.6 Uses of Raspberry Pi

There are so many things you can make with a Raspberry Pi. And due to its extreme popularity, the number of third-party sensors, modules, and code libraries has grown to epic proportions. This has greatly expanded the possible applications for the Raspberry Pi. The applications really are endless, but here are some common ones:

- Web servers
- Local network hubs for IoT
- Sensor control Sensor control
- Robotics control
- Industrial equipment control
- Automated control systems
- Data processing

2.7 Programming the Raspberry Pi

One of the goals of the creators of the Raspberry Pi was to create a computing platform that would allow people with no background in computer science or programming to learn how to use and program a computer. This is one of the reasons why the Raspberry Pi supports a wide range of programming languages:

- Python
- Java
- C/C++
- HTML5
- Scratch
- JavaScript
- jQuery
- Perl
- Erlang

The Raspberry Pi is an amazing feat of engineering and anyone interested in programming or building electronics projects should consider learning how to use it. There are hundreds of microcontroller and development boards out there, but the Raspberry Pi combines all of those functions on a device the size of a credit card.

CHAPTER 3:
SENSORS & ACTUATORS

The Systems relies on a set of sensors and actuators to detect and analyze information about the surrounding vehicles and prevent collisions between them.
These sensors and actuators are installed in the car and work together to provide a detailed picture of the environment around the vehicle.
Among the sensors that can be used are:

3.1 INFRARED SENSOR

3.1.1 Introduction

The IR Sensor HW-201 is a reflective optical sensor used for short-range object detection, line following, and obstacle sensing applications. It works by emitting infrared light and detecting the intensity of the reflected light from nearby surfaces. When an object passes in front of the sensor, it reflects the IR light back to the photodiode, which results in a change in voltage that can be detected by a microcontroller. This sensor operates within a short detection range, typically from 2 cm to 30 cm, depending on the surface color and ambient light conditions.

The IR sensor consists of an IR LED and a photodiode arranged in such a way that the emitted infrared light reflects off a surface and is captured by the photodiode. The amount of reflected IR light is directly related to the proximity of an object and the reflectivity of its surface. White or light-colored surfaces reflect more IR light, making them easier to detect, whereas dark or black surfaces absorb more light, resulting in weaker signals.

3.1.2 About the Module

The HW-201 module integrates the following components:

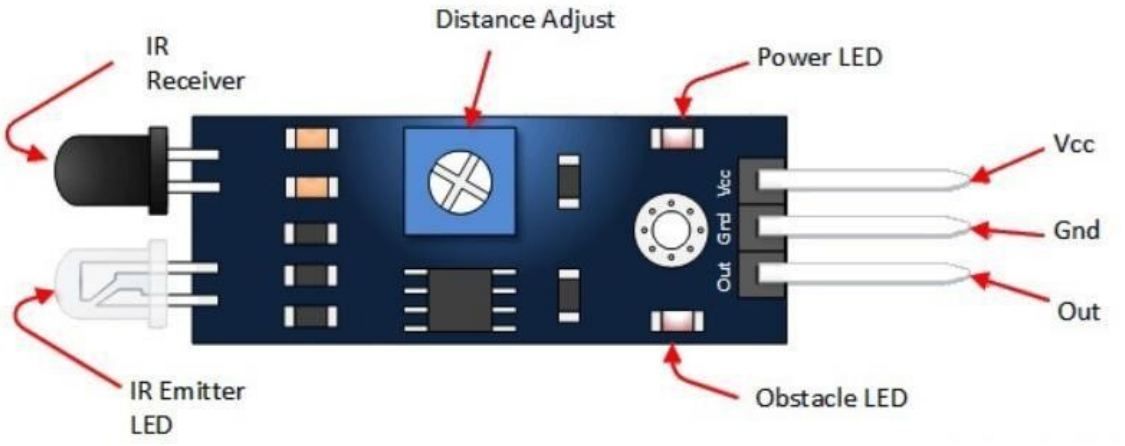


Figure 3.1

- IR Emitter LED (Transmitter)
  - Located at the front of the module.
  - Continuously emits infrared light forward.
  - White in color in the image.
- IR Receiver (Photodiode or Phototransistor) ○ Positioned beside the IR LED.
  - Detects the amount of reflected IR light from nearby objects.
  - Black in color in the image.
- Comparator IC (LM393)
  - Located at the center of the board.
  - Compares the signal received from the IR receiver with a reference voltage.
  - Converts the analog signal into a digital output based on reflectivity.
- Distance Adjustment Potentiometer
  - Blue knob near the center.
  - Used to tune the sensitivity and detection distance of the module.
  - Rotating the knob adjusts the threshold of detection.
- Power LED
  - Indicates when the module is powered.

    - Located near the VCC pin and glows red when powered.
- Obstacle LED
    - Lights up when an object is detected.
    - Indicates detection by glowing (usually green or red).

The module outputs a digital HIGH or LOW signal depending on the presence of an object. It features a simple 3-pin interface:

- VCC: Power supply input (typically 3.3V to 5V DC)
- GND: Ground connection
- OUT: Digital output signal to the microcontroller (LOW when object is detected, HIGH otherwise)

Figure 3.2

3.1.3 HW-201 Specifications

These specifications are based on typical datasheets and supplier documentation:

- Operating Voltage: 3.3V – 5V DC
- Detection Distance: 2 – 30 cm (adjustable)
- Output Type: Digital (Active LOW)
- Adjustable Sensitivity: Yes, via onboard potentiometer
- Response Time: < 1 ms
- Dimensions: Approx. 32mm x 14mm x 8mm
- Weight: Approx. 4g

3.1.4 How do IR Sensors work?

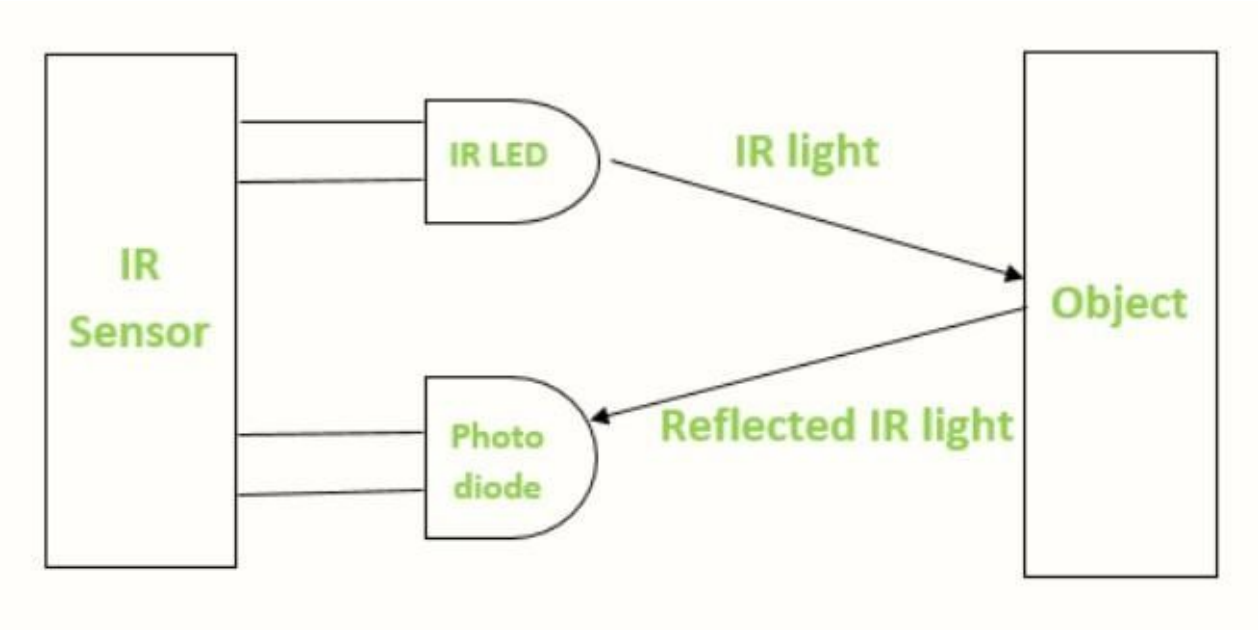


Figure 3.3

The IR HW-201 module continuously emits infrared light. When an object is placed in front of the sensor within its detection range, the IR light reflects back toward the module and is captured by the photodiode. The comparator processes this signal and compares it to a reference voltage defined by the potentiometer. If the reflected light intensity exceeds this threshold, the module pulls the output pin LOW, signaling that an object is present. Otherwise, it remains HIGH. This simple mechanism makes the HW-201 ideal for applications such as line-following robots, presence detection, or object counters in embedded systems.

3.1.5 IR Usage in our Project?

In our Lane Departure Warning (LDW) system, the IR Sensor HW-201 plays a critical role in detecting the presence of lane markings on the road surface. We use multiple HW-201 modules mounted underneath the vehicle or aligned near the wheels to continuously monitor the surface below. When the vehicle deviates from its lane, the sensor detects a change in reflectivity between the road and the lane line—typically a difference in color or brightness (e.g., black road and white line).

The IR emitter emits infrared light toward the ground, and the receiver monitors the reflected signal. A high level of reflection (such as from white or light-colored lane markings) causes the sensor to trigger a LOW output, which is then captured by the STM32 microcontroller using an external interrupt. This change allows our system to determine if the car has crossed a line unintentionally. In response, an alert signal is activated via a buzzer module to warn the driver.

3.2 Buzzer

3.2.1 Introduction

- An audio signaling device like a beeper or buzzer may be electromechanical or piezoelectric or mechanical.
- The main function of this is to convert the signal from audio to sound.
- Generally, it is powered through DC voltage and used in timers, alarm devices, printers, alarms, computers, etc.
- Based on the various designs, it can generate different sounds like alarm, music, bell & siren.

3.2.2 Buzzer Pin Configuration

The pin configuration of the buzzer is shown below. It includes two pins namely positive and negative. The positive terminal of this is represented with the '+' symbol or a longer terminal. This terminal is powered through 6 Volts whereas the negative terminal is represented with the '-' symbol or short terminal and it is connected to the GND terminal.

Figure 3:4

3.2.3 Specifications

- The frequency range is 3,300Hz.
- Operating Temperature ranges from – 20° C to +60°C.
- Operating voltage ranges from 3V to 24V DC.
- The sound pressure level is 85dBA or 10cm.
- The supply current is below 15mA

3.2.4 Working principle:

The working principle of a buzzer is based on the piezoelectric effect.

- A piezoelectric material is one that generates an electric charge in response to mechanical stress and produces mechanical stress in response to an electric field. Buzzer sensors typically use a piezoelectric ceramic element as the sound-producing element.
- When a voltage is applied across the piezoelectric element, it causes the element to vibrate at a high frequency, typically in the range of several kilohertz to several megahertz.
- These vibrations cause the air molecules around the element to vibrate, producing sound waves that propagate through the air.
- The frequency of the sound produced by the buzzer is determined by the frequency of the voltage applied to the piezoelectric element.

- The intensity of the sound, or the volume, is determined by the amplitude of the voltage.
- Buzzer sensors can be designed to produce different types of sounds, including continuous tones, intermittent tones, and complex tone patterns.
- The specific sound produced by the buzzer is determined by the design of the piezoelectric element and the electrical circuit driving it.

Overall, the working principle of a buzzer is based on the piezoelectric effect and the vibration of a piezoelectric element to produce sound waves in the surrounding air.

3.3 DC Motor

3.3.1 Introduction

A DC motor is an electrical motor that uses direct current (DC) to produce mechanical force. The most common types rely on magnetic forces produced by currents in the coils. Nearly all types of DC motors have some internal mechanism, either electromechanical or electronic, to periodically change the direction of current in part of the motor.

DC motors were the first form of motors widely used, as they could be powered from existing direct-current lighting power distribution systems. A DC motor's speed can be controlled over a wide range, using either a variable supply voltage or by changing the strength of current in its field windings. Small DC motors are used in tools, toys, and appliances. The universal motor, a lightweight brushed motor used for portable power tools and appliances can operate on direct current and alternating current. Larger DC motors are currently used in propulsion of electric vehicles, elevators and hoists, and in drives for steel rolling mills. The advent of power electronics has made replacement of DC motors with AC motors possible in many applications

Figure 3:5

3.3.2 Basic Principles and Operation

At the heart of a DC motor's operation is the interaction between magnetic fields and current carrying conductors. The basic principle involves generating rotational motion through the Lorentz force, which acts on a conductor carrying current within a magnetic field.

Components of a DC Motor:

- Stator: The stationary part of the motor, which produces a magnetic field.
- Rotor (Armature): The rotating part that carries the current and interacts with the stator's magnetic field.
- Commutator: A rotary switch in brushed DC motors that reverses the direction of current through the rotor windings.
- Brushes: Conductive materials that maintain an electrical connection between the stationary and rotating parts of the motor.

When current flows through the rotor windings, a magnetic field is created around the rotor. This

The field interacts with the stator's magnetic field, producing a torque that causes the rotor to turn. The commutator periodically reverses the current direction, ensuring continuous rotation.

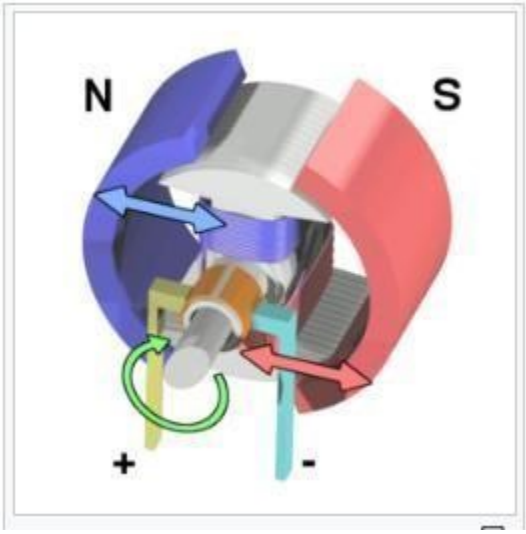


Figure 3:6

3.3.3 Types of DC Motors

- **Brushed DC Motors:**

  Brushed DC motors are the most traditional type of DC motor, featuring a simple design where the commutator and brushes manage the current direction.

  0 **Structure and Function:** Consists of a rotor with windings, a commutator, and brushes that make contact with the commutator segments.

  ○ **Advantages:** Simple control, low cost, and easy to maintain.

  ○ **Disadvantages:** Brush wear and tear, frequent maintenance, and limited speed control precision.

- **Brushless DC Motors:**

  Brushless DC motors (BLDC) eliminate the commutator and brushes, using electronic

  controllers to manage current flow.

  0 **Structure and Function:** Comprises a rotor with permanent magnets and a stator with windings, controlled by an electronic controller.

  ○ **Advantages:** Higher efficiency, less maintenance, and better speed and position control.

  ○ **Disadvantages:** Higher initial cost and more complex control system

  .

- **Applications:**
    0 **Brushed DC Motors:** Suitable for applications where simplicity and cost effectiveness are crucial, such as in toys, small appliances, and some power tools.
    - **Brushless DC Motors:** Ideal for applications requiring high efficiency and precise control, including drones, electric vehicles, and industrial robots.

3.3.4 Detailed Components and Design

- **Stator Design and Materials**

  The stator is typically made of laminated steel to minimize eddy current losses, with windings made from copper or aluminum.

- **Rotor Design and Materials**

  The rotor is constructed from laminated steel and wound with copper wire, designed to handle high rotational speeds and torque.

- **Commutator and Brushes**

  In brushed motors, the commutator is made of segmented copper rings, and brushes are often made of carbon or graphite, chosen for their conductivity and wear resistance.

- **Electronic Controllers**

  In brushless motors, electronic controllers manage the current flow through the stator windings, using sensors to determine the rotor's position and adjust the current accordingly.

3.3.5 Performance Characteristics

- **Torque-Speed Characteristics :**

  DC motors exhibit a linear relationship between torque and speed, making them suitable for applications requiring a wide range of speeds and consistent torque.

- **Efficiency Factors:**

  Efficiency is influenced by factors such as winding resistance, friction, and magnetic losses.

  Brushless motors generally offer higher efficiency due to reduced friction and more precise control.

- **Common Performance Metrics:**

  Key performance metrics for DC motors include torque, speed, power output, and efficiency. These metrics help determine the suitability of a motor for specific applications.

3.4 H-bridge

3.4.1 Introduction

The H-bridge is a fundamental circuit in electrical engineering, widely used for controlling the direction of a DC motor. It allows for the application of voltage across a load in either direction, making it essential for various applications such as robotics, automotive systems, and industrial machinery. The H-bridge's ability to control motor direction and speed efficiently makes it a critical component in modern electronic design. This section provides a brief overview of the Hbridge, its importance, and its key applications.

3.4.2 Basic Principles

An H-bridge is a type of electrical circuit that enables a voltage to be applied across a load in either

direction. This circuit is named for its distinctive 'H' shape, formed by four switching elements, usually transistors or MOSFETs, arranged in a configuration that allows for the control of current flow through a load, such as a DC motor.

3.4.3 Components of an H-Bridge

- **Switching Elements (Transistors/MOSFETs):**

  The primary components of an H-bridge are four switches, which can be implemented using bipolar junction transistors (BJTs), metal-oxide semiconductor field-effect transistors (MOSFETs), or even relays.

  These switches are labeled as Q1, Q2, Q3, and Q4, with Q1 and Q3 on one side of the load and Q2 and Q4 on the other.

- **Diodes:**

  Diodes are often used in H-bridge circuits to protect against back electromotive force (EMF) generated by inductive loads like motors.

These are known as flyback diodes.

They ensure that the current flows in the correct direction and help in managing transient voltage spikes.

- **Control Logic:**

  The operation of the H-bridge is controlled by a logic circuit that determines which switches are turned on or off at any given time.

  This logic can be implemented using microcontrollers, dedicated motor driver Ics, or discrete logic gates.

The arrangement of these components in an h-bridge allows for the efficient control of the load's direction and in more advanced designs, the speed of the load.

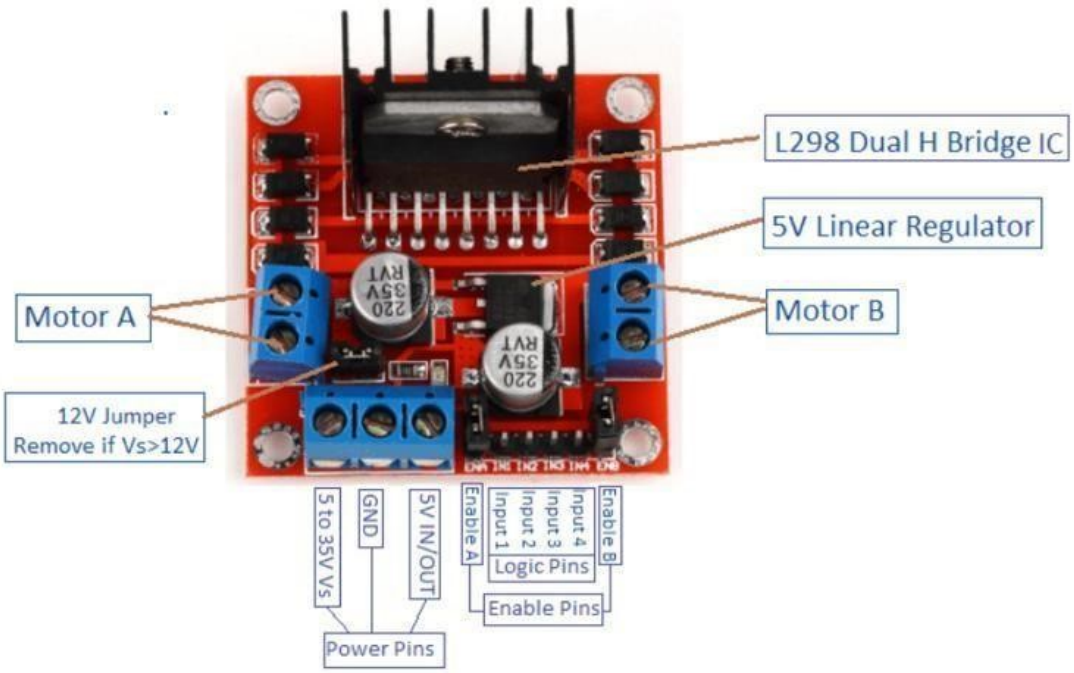


Figure 3.7

3.4.4 Operation

The operation of an H-bridge involves switching the transistors in various combinations to control the direction and flow of current through the load.The four basic states of an H-bridge are:

\

- **Forward Operation:**

  In forward operation, switches Q1 and Q4 are closed (turned on), while Q2 and Q3 are open (turned off). This allows current to flow from the power supply, through Q1, the load, Q4, and then to ground.
  This configuration drives the motor in one direction.
- **Reverse Operation:**

  In reverse operation, switches Q2 and Q3 are closed, while Q1 and Q4 are open. This causes current to flow in the opposite direction through the load.This reverses the motor's direction.
- **Braking Mode:**
  To brake the motor, both Q1 and Q2, or Q3 and Q4, can be closed simultaneously. This short-

  circuits the motor's terminals, causing it to stop quickly due to the back EMF.
- **High-Impedance State:**
  In the high-impedance state, all switches are open, and no current flows through the motor. This effectively disconnects the motor from the power supply, allowing it to freewheel.

The control logic must ensure that no two switches on the same side of the load (e.g., Q1 and Q2) are closed simultaneously, as this would create a short circuit directly across the power supply.

3.5 Infrared Night Vision

3.5.1 Introduction

The Raspberry Pi Infrared Night Vision Camera (5MP - 160° FOV) is a high-resolution imaging module equipped with infrared (IR) capability, designed for both day and night surveillance and vision-based applications. Built around the 5-megapixel OV5647 image sensor, this camera delivers sharp and clear images with a wide-angle 160-degree field of view, making it suitable for robotics, wildlife monitoring, and embedded vision systems. What sets this module apart is its

integrated night vision support, which allows it to capture images in complete darkness using IR LEDs. This is achieved by removing the standard IR-cut filter found in typical cameras, allowing infrared light to reach the sensor.

3.5.2 About the Module

The Night Vision Camera module includes several key components:

- **OV5647 5MP Sensor**

  Provides a resolution of 2592x1944 pixels and supports 1080p30, 720p60, and 640x480p60/90 video output.

- **Wide-Angle Lens (160° FOV)**

  Mounted with a fisheye lens that allows for an ultra-wide viewing area, ideal for full-scene monitoring.

- **IR LED Boards (x2)**

  Mounted on either side of the camera module. These 850nm infrared LEDs illuminate dark environments, enabling night vision capability.

- **Light-Dependent Resistor (LDR) & Control Circuit**

  Automatically turns on the IR LEDs in low-light conditions. The LDR detects ambient light levels, allowing the module to operate hands-free during night transitions.

- **Standard CSI Interface (15-pin)**

  Connects directly to the CSI camera port of the Raspberry Pi boards for high-speed video and image data transmission.

- **Mounting Holes and Adjustable Focus**

  Easy to fix in enclosures or project setups, and the lens can be manually adjusted for clear focus based on application needs.

Figure 3.8

3.5.3 Specifications

- Image Sensor: 1/4” OV5647 CMOS
- Resolution: 5MP (2592×1944), supports up to 1080p video
- IR Support: Yes, with dual 850nm IR LEDs
- Field of View: 160°
- Interface: CSI (Camera Serial Interface)
- Operating Voltage: 3.3V via CSI interface
- LED Trigger: Automatic via onboard LDR
- Dimensions: ~25mm x 24mm (camera board only)
- Lens: Manual focus, adjustable

3.5.4 How the Night Vision Camera Works

During normal lighting conditions, the camera captures visible light like any standard module. In low-light or dark conditions, the LDR triggers the IR LEDs, which emit 850nm infrared light that is invisible to the human eye but detectable by the camera sensor. Since the camera lacks an IR-cut filter, it can pick up this light and form a complete image even in total darkness. The IR illumination reflects off objects in the scene and returns to the sensor, enabling consistent imaging performance 24/7. This mechanism is highly effective for motion detection, surveillance, and dark environment automation tasks.

3.5.5 Night Vision Camera Usage in Our Project

In our project, the Raspberry Pi Infrared Night Vision Camera is used as part of the vision system. The camera enables the system to continuously monitor the road environment under varying lighting conditions—including complete darkness—by capturing clear, high-resolution images using infrared illumination.

This camera is particularly useful at night or in low-light areas where traditional vision systems would struggle. With its wide 160° field of view and built-in IR LEDs, the camera can detect lane markings, vehicles, pedestrians, and other obstacles even when ambient light is minimal. This enhances the vision system's reliability, making it capable of supporting real-time decisions based on visual inputs.

Our project uses this camera to feed image data into processing algorithms that analyze road conditions and detect potential hazards. The system then responds with appropriate warnings or actions, such as triggering a buzzer or logging an event.

By integrating the Raspberry Pi Night Vision Camera into our vision module, we ensure that the vehicle maintains visual awareness of its surroundings at all times—an essential feature for any intelligent safety system inspired by ADAS technologies.

3.6 Bluetooth Module HC-05

3.6.1 Introduction

The HC-05 Bluetooth module is a popular, easy-to-use, and cost-effective wireless communication module designed for short-range data transmission between embedded systems and mobile devices. It supports Bluetooth Serial Port Profile (SPP), which enables it to replace RS232 wired serial connections. It is widely used in robotics, home automation, vehicle communication systems, and more.

The HC-05 can act as both master and slave, allowing it to initiate or respond to Bluetooth connections. It operates on the 2.4GHz ISM band and provides a reliable, low-power wireless link with a range of approximately 10 meters.

3.6.2 About the Module

The HC-05 Bluetooth module includes the following components:

- **Bluetooth Chip (CSR BlueCore4/BC417)**
  The heart of the module responsible for handling Bluetooth communication.
- **3.3V Regulator**
  Onboard voltage regulator that steps down 5V input to 3.3V required by the chip.
- **LED Indicator**
  Indicates the current operating state (e.g., pairing, connected, etc.).
- **EN Pin (Enable)**
  Activates the module when pulled high.
- **STATE Pin**
  Used to monitor the current connection status.
- **Key/Mode Pin**
  Used to switch between command and data modes.
- **RX and TX Pins**
  Serial data input and output (UART). Used to connect the module with microcontrollers such as STM32.
- **VCC and GND**
  Power supply pins. VCC typically supports 3.6 -- 6V, although the logic level is 3.3V.

3.6.3 Specifications

- Bluetooth Version: 2.0 + EDR (Enhanced Data Rate)
- Operating Frequency: 2.4 GHz ISM Band
- Communication Protocol: UART (Serial Port Profile - SPP) ● Master/Slave Capability: Supports both roles (configurable)
- Default Baud Rate: 9600 bps (configurable via AT commands: 9600–115200 bps)
- Operating Voltage: 3.6V to 6V (3.3V logic level)
- Current Consumption: 30–40 mA (active), ~8 mA (idle)
- Range: Up to 10 meters (unobstructed) ● Dimensions: ~28 mm x 15 mm x 2.35 mm

3.6.4 How the Bluetooth Module Works

The HC-05 module works by establishing a UART-based serial communication channel over Bluetooth. In master mode, it can actively search for and connect to other Bluetooth devices. In slave mode, it waits for pairing from a master device such as a smartphone or Raspberry Pi. Once paired, it transmits and receives serial data over the air, just like a wired serial port. It can be configured using AT commands to set parameters such as baud rate, module name, PIN, and master/slave role.

When connected to a microcontroller (e.g., STM32F103C8T6), the TX and RX pins are connected to USART lines, allowing the system to send/receive wireless messages, such as control commands or sensor data, in real-time.

Figure 3.9

CHAPTER 4:
The Design

4.1 Introduction

In any project ,the most important thing that may cause a lot of problems and it should take the most attention is the design of the body for the project.
Taking into consideration that we have designed the car as a miniature example of real cars integrated with several systems, the design will be discussed and explained in detail, starting with the design program used.

4.2 Design Program Used

The program is called “solidwork”, **SolidWorks** is a **computer-aided design (CAD)** and **computer-aided engineering (CAE)** software developed by Dassault Systèmes. It is widely used for 3D modeling, mechanical design, and product simulation. Here is a clearly defined description :

1. **SolidWorks** is a **parametric solid modeling software** used mainly in mechanical and product design.

2. It allows engineers and designers to create precise **3D models** and **2D drawings** of parts and assemblies.

3. It features tools for **sketching, part modeling, assembly design, and sheet metal** design.

4. SolidWorks uses a **feature-based** approach, where models are built by applying features like extrude, cut, fillet, etc. It provides **simulation and analysis tools** for stress, motion, fluid flow, and thermal conditions.

5. It supports **design automation** through equations and design tables.

6. SolidWorks includes a **materials library** to simulate real-world properties like density and strength.

7. It integrates well with **PDM (Product Data Management)** systems for version control and collaboration.

8. The **user interface** is intuitive, with a feature tree for easy editing and history tracking.

9. It allows **importing/exporting** various CAD file formats like STEP, IGES, STL, and DXF.

10. SolidWorks supports **add-ins** like Simulation, Electrical, and CAM for specialized tasks.

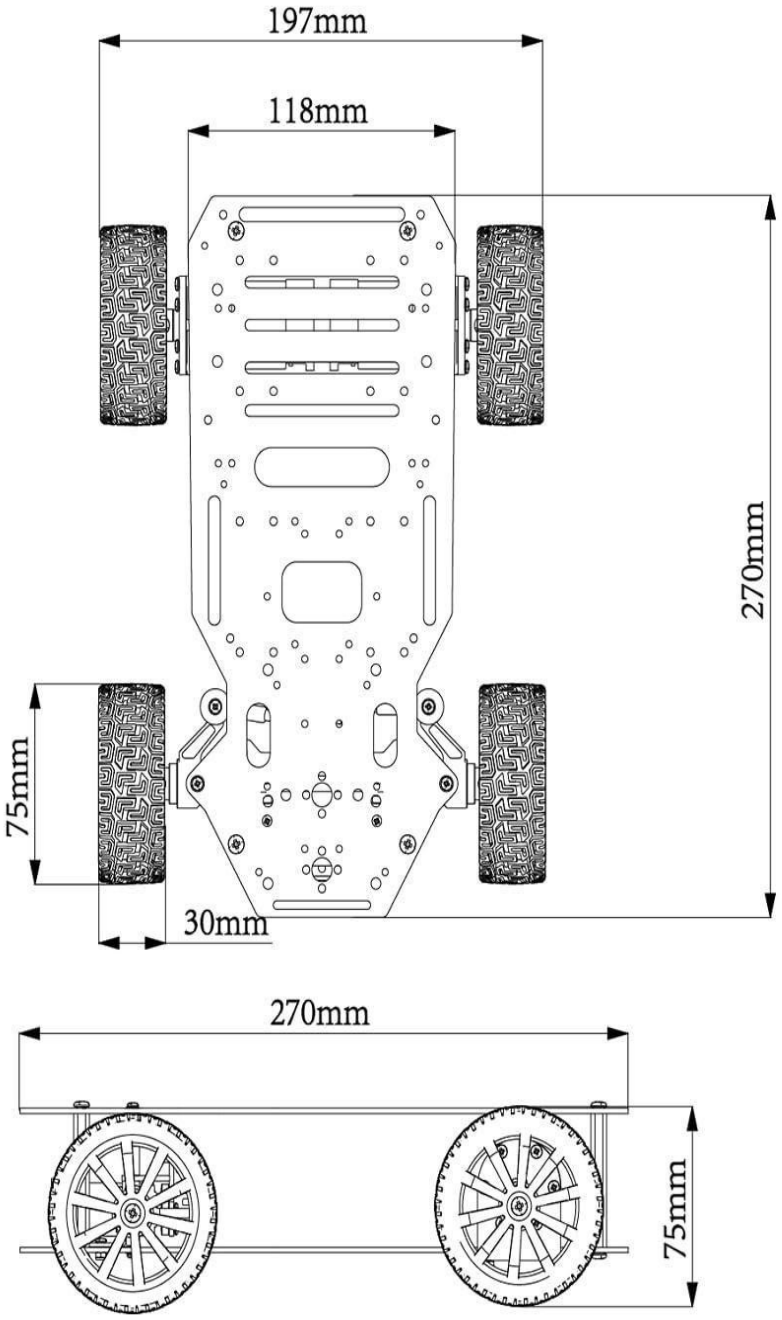
11. It is commonly used in industries such as automotive, aerospace, consumer products, and manufacturing.

12. The software offers **3D rendering and animation** capabilities for product presentations.

13. SolidWorks enhances productivity by enabling **rapid prototyping and virtual testing** before physical production.

4.3 The Design Overview

Next, we will see the overview of the car design and its dimensions all that shows in figure 4.1



4.1 Design overview

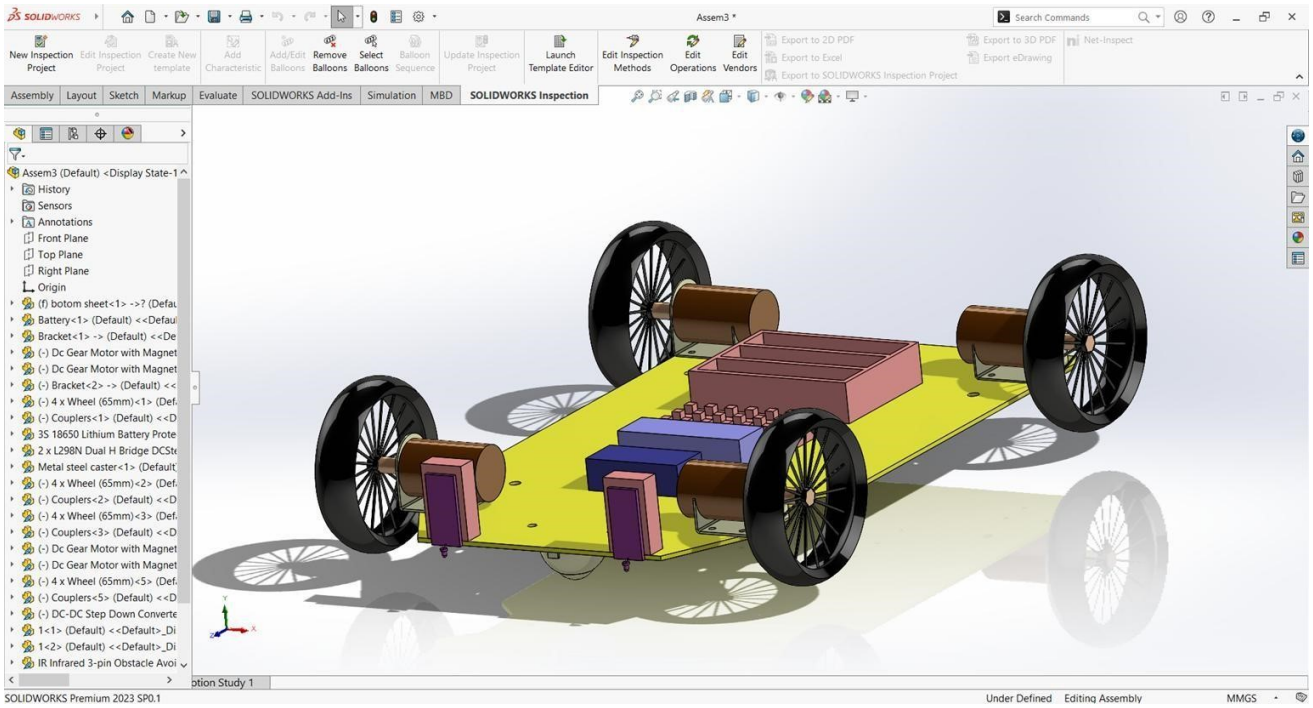


figure 4.2

In figure 4.1 , we can see that first floor has the major parts of hardware which is the following :

1. 2 back wheels
2. 2 motors , connected to 2 back wheels
3. 2 IR sensors for LDW system
4. 3 of li-ion batteries (3.7V /4.2V – 6800mAh) with a holder
5. BMS (Battery Management System)
6. DC-DC Step Down Converter (4.5–40V to 5V/2A USB)
7. Adjustable DC-DC Step Down Converter (4.5–60V to 3–35V, 3A
8. BreadBoard
9. STM32f301c8t6
10. Bluetooth modules are connected by jumpers or 14 AWG together .

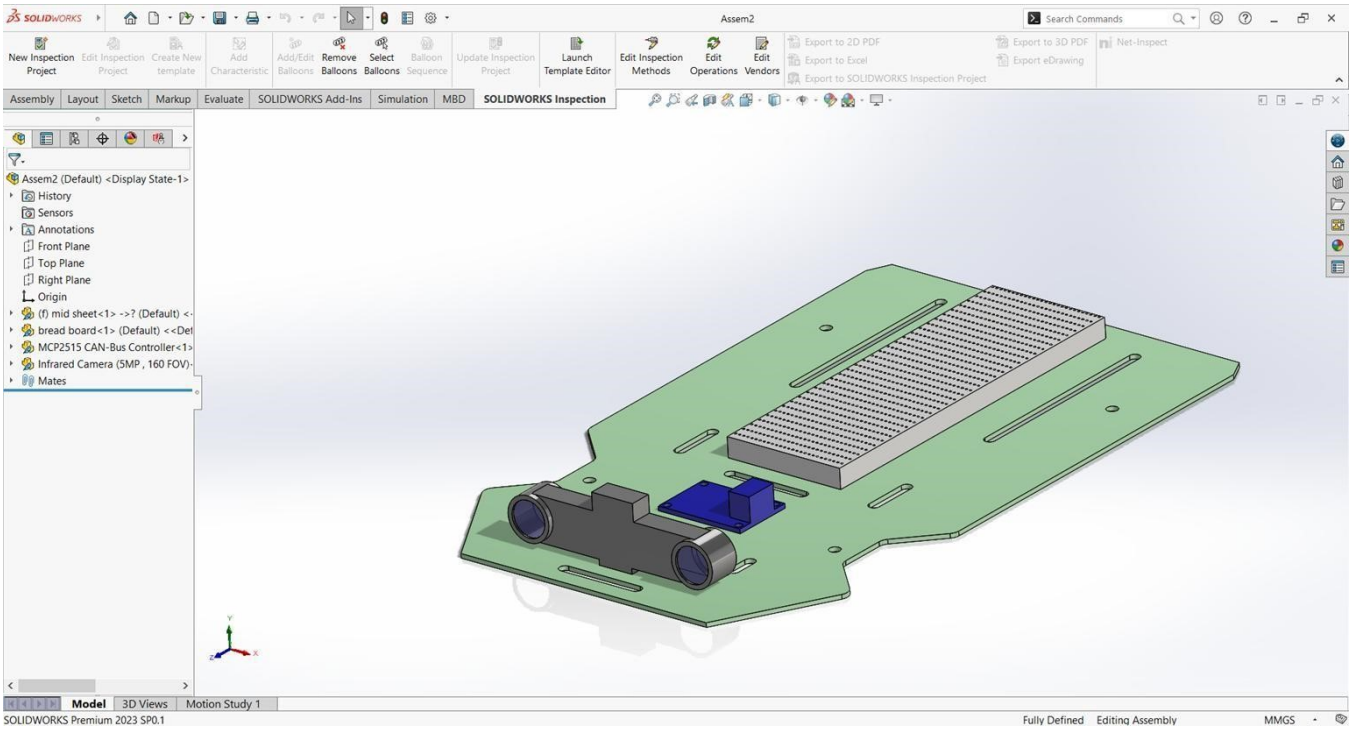


figure 4.3

As we can see in figure 4.2 , the second floor of the car has the most important part of the hardware which is the following :

1. Raspberry PI
2. Display screen
3. Powerbank for raspberry pi
4. Infrared camera

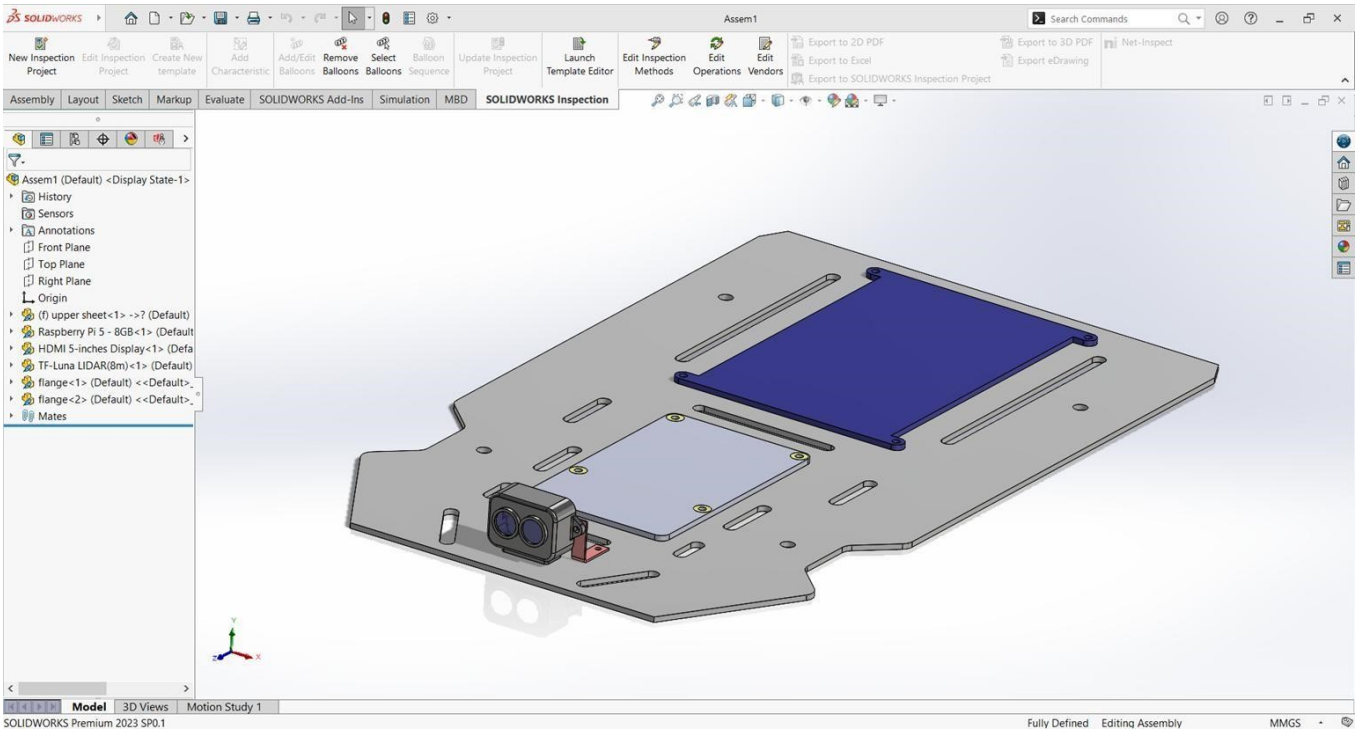


figure 4.4

As we can see in figure 4.3 , This cover is designed to cover the car and enhance its appearance. It does not contain any hardware parts, but rather holds them and applies an outer layer.

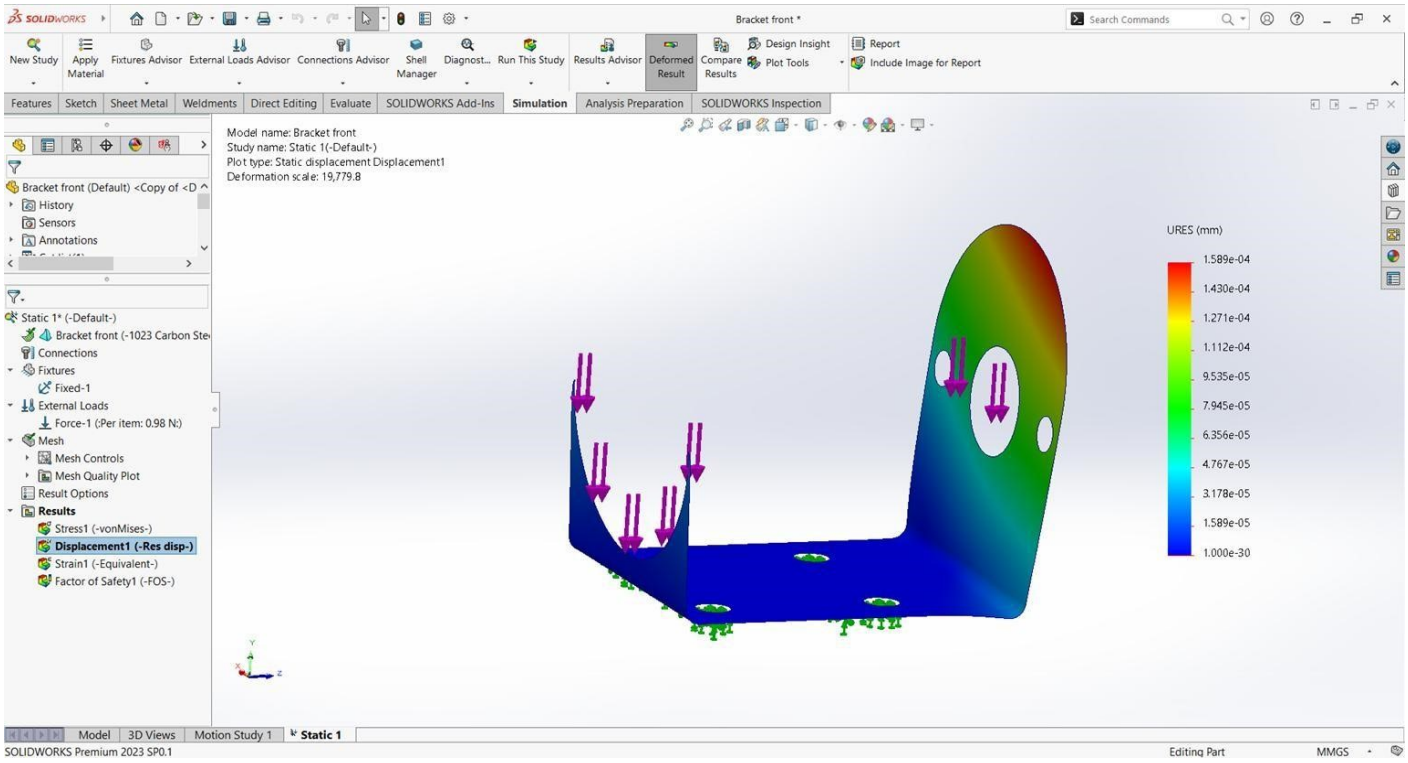


figure 4,5

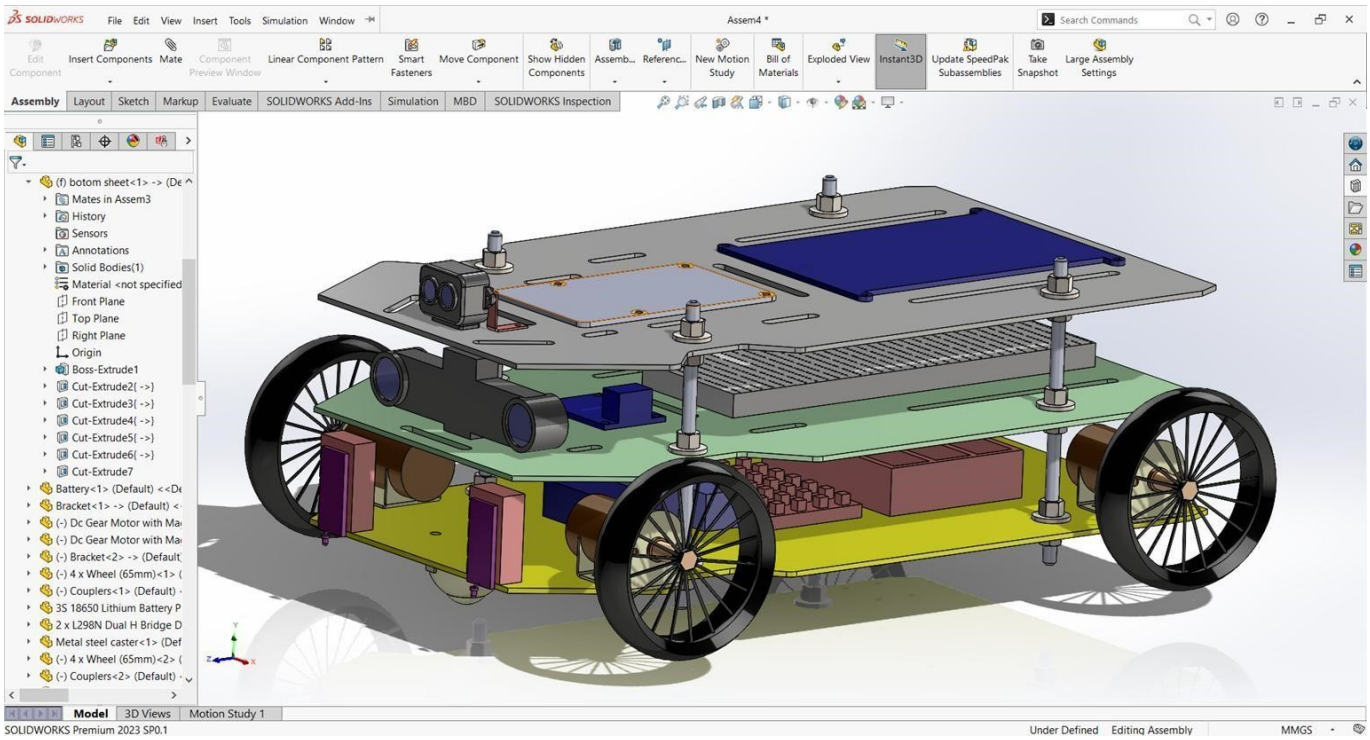


figure 4.6

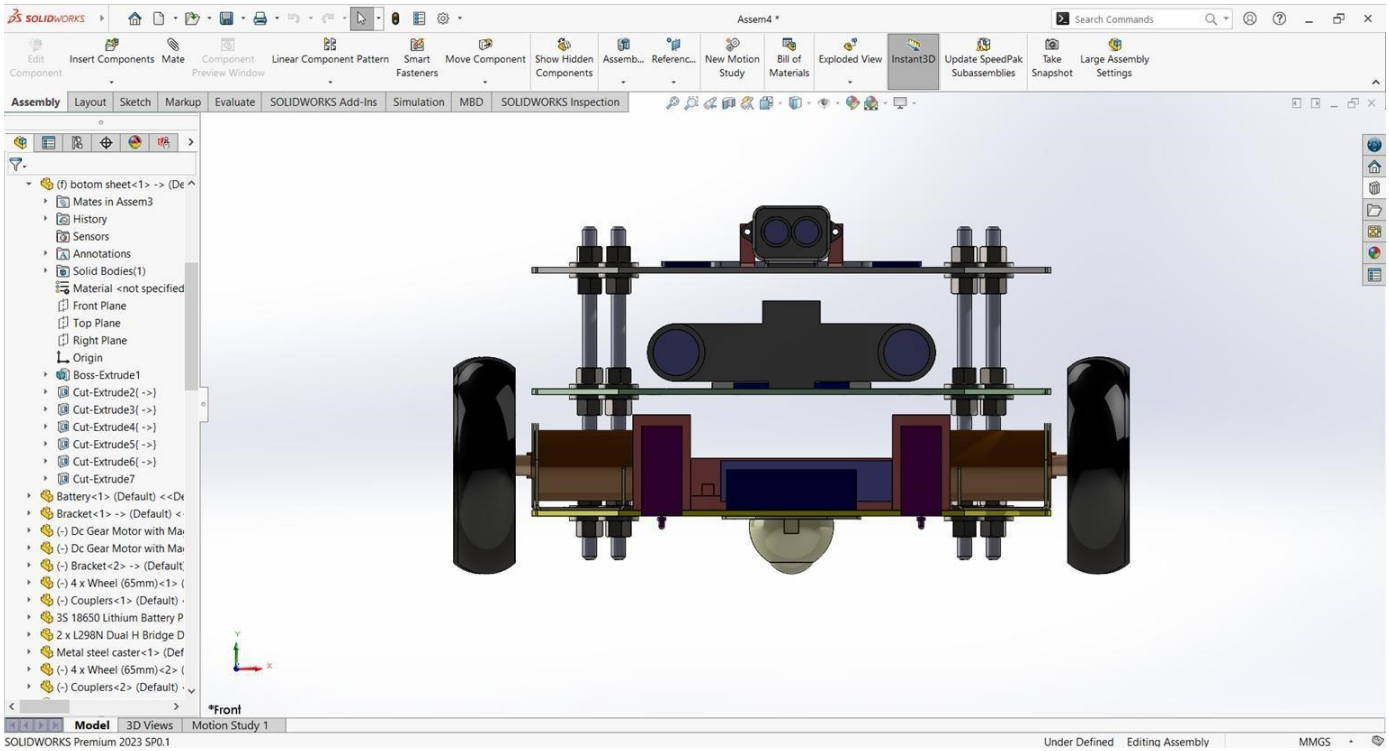


figure 4.7

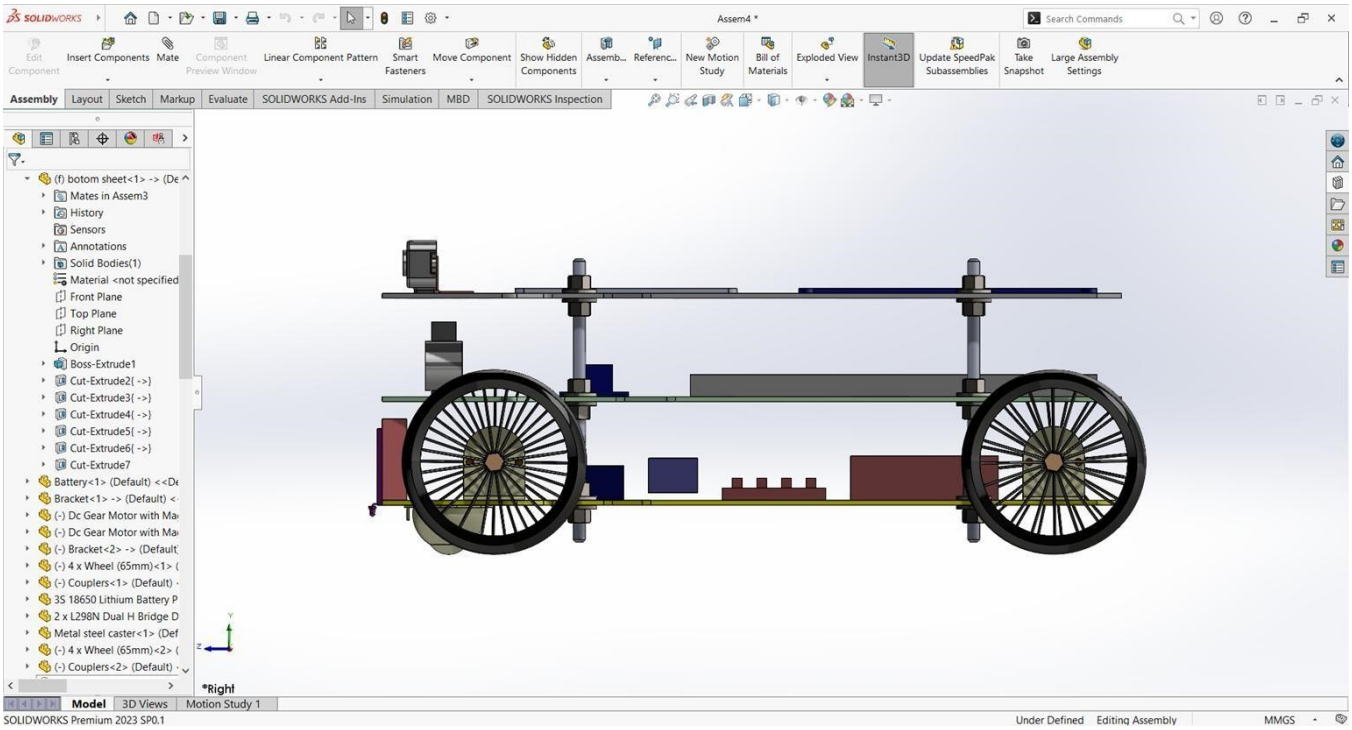


figure 4.8

As we can see in figures 4.6 ,4.7 and 4.8 , all of them shows how the car will be in real life after made

Note that the two front wheels can be dispensed with and one wheel can be placed in the middle of the distance between them.

Module B

Software Architecture

Chapter 5 : Controller Area Network (CAN)

Chapter 6 : D-Bus.

CHAPTER 5:
Controller Area Network (CAN)

5.1 Introduction

Before 1985, automotive wiring systems were extremely complex, bulky, and costly. Each Electronic Control Unit (ECU) in the vehicle needed to be connected to other components directly using point-to-point wiring. As the number of ECUs increased — handling functions like engine control, braking, airbags, lighting, and more — the wiring harness became heavier, more expensive, and harder to maintain. This created a pressing need for a more efficient and standardized method of communication between these units.

To solve this challenge, Bosch developed the Controller Area Network (CAN) protocol in 1983 and released it in 1986. CAN is a high-integrity, serial communication bus designed to enable real-time data exchange between multiple intelligent devices (nodes) in a vehicle using only two twisted-pair wires (CAN High and CAN Low). This reduced both the weight and complexity of the wiring, while also improving communication speed and reliability.

Due to its robustness, error-handling capability, and deterministic behavior, CAN quickly became the industry standard. By 1993, CAN was officially standardized as ISO 11898, establishing it as the default in-vehicle network protocol.

In 1996, the United States mandated the integration of CAN as part of the On-Board Diagnostics version II (OBD-II) for all cars and light trucks. This regulation required manufacturers to use CAN to report diagnostic and emissions-related data, reinforcing its position as a cornerstone in modern vehicle communication systems.

Today, CAN is not only widely adopted in the automotive industry but also used in other fields such as industrial automation, medical equipment, and robotics. In embedded system applications like our project, we utilize CAN with the STM32F103C8T6 (Blue Pill) microcontroller, which includes a built-in CAN controller, making it a cost-effective and efficient platform for implementing a real-time, multi-node communication system in our smart car prototype.

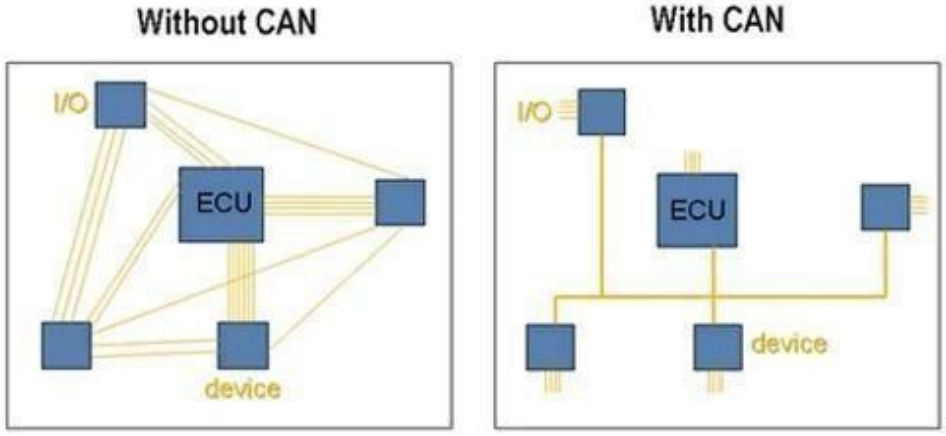


Figure 5.1

5.2 What is CAN?

CAN, short for Controller Area Network, is a robust serial communication protocol originally developed by Bosch and now defined by the international standard ISO 11898. It was specifically designed to enable reliable, real-time communication between multiple electronic control units (ECUs) within vehicles and other embedded systems. The collective set of rules governing communication, wiring, and data formatting under CAN is commonly referred to as the CAN bus.[16]

Unlike traditional serial protocols such as UART or SPI, which are typically point-to-point, the CAN protocol is multi-master and broadcast-based. This means that multiple nodes (devices) can communicate over the same two-wire differential bus, where every node can transmit or receive data without the need for a centralized controller. All nodes on the bus receive all messages, but each node can be configured to filter and respond only to specific message identifiers.

The CAN protocol defines:

- How the physical bus is constructed (usually using twisted-pair wires labeled CAN_H and CAN_L)
- How messages are framed, including identifiers, data length, CRC error-checking, and acknowledgment bits
- How nodes gain access to the bus, using a non-destructive arbitration method based on message priority
- How errors are detected and managed, including automatic retransmission and fault confinement

5.3 Features of CAN protocol

- Low cost: Since a CAN serial bus uses two wires (with high volume and low cost production), it offers a good price-to-performance ratio.
- Bit-rate: the bit rate is uniform and fixed for all the nodes and the speed of the CAN may be different for different networks available in a system.
- System flexibility: it is easy for engineers to integrate new electronic devices into the CAN bus network without significant programming overhead and supports a modular system that is easily modified to suit your specs or requirements.
- Message routing: every message is identified by a special unique identifier that does not indicate the destination of the message but only describes the meaning of the data available in the message. So that all the nodes connected in the network can decide by message filtering technology using this message ID whether to receive the data or not.
- Multi-master communication: Any node can access the bus.
- Multicast: more than one node/ECU in the network is able to receive the same transmitted message.
- Arbitration: If two or more nodes start transmitting messages at the same time, the bus access conflict is resolved by the bitwise arbitration using the Identifier. The mechanism of the arbitration guarantees that neither information nor time is

lost. If a data frame and a Remote frame with the same Identifier are initiated at the same time, the Data frame prevails over the Remote frame.

- Priorities: every message has a priority, so if two nodes try to send messages simultaneously, the one with the higher priority gets transmitted and the one with the lower priority gets postponed. This arbitration is non-destructive and results in the non-interrupted transmission of the highest priority message.
- Error Capabilities: the CAN specification includes a Cyclic Redundancy Code (CRC) to perform error checking on each frame's contents. Frames with errors are disregarded by all nodes, and an error frame can be transmitted to signal the error to the network. Global and local errors are differentiated by the controller, and if too many errors are detected, individual nodes can stop transmitting errors or disconnect themselves from the network completely.

5.4 Can Network Message Format

CAN devices send data across the CAN network in packets called frames.

These frames can be differentiated based on identifier fields. A CAN frame with 11-bit identifier fields is called Standard CAN and with a 29-bit identifier, fields called the extended frame

1. Standard frame:

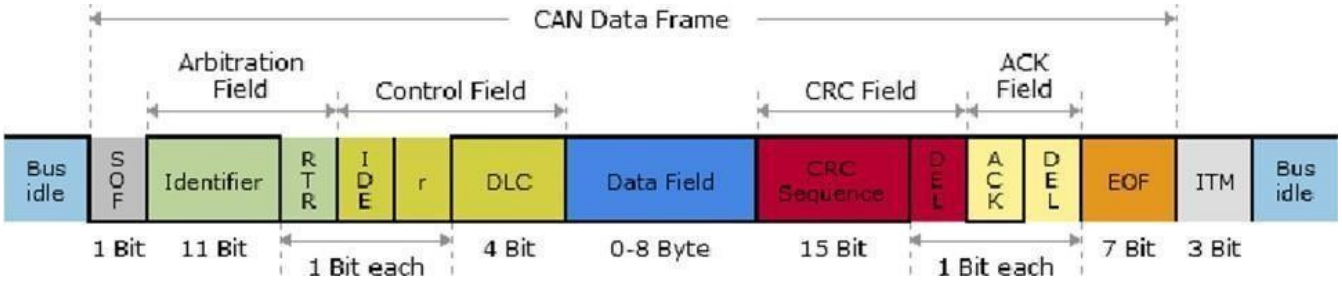


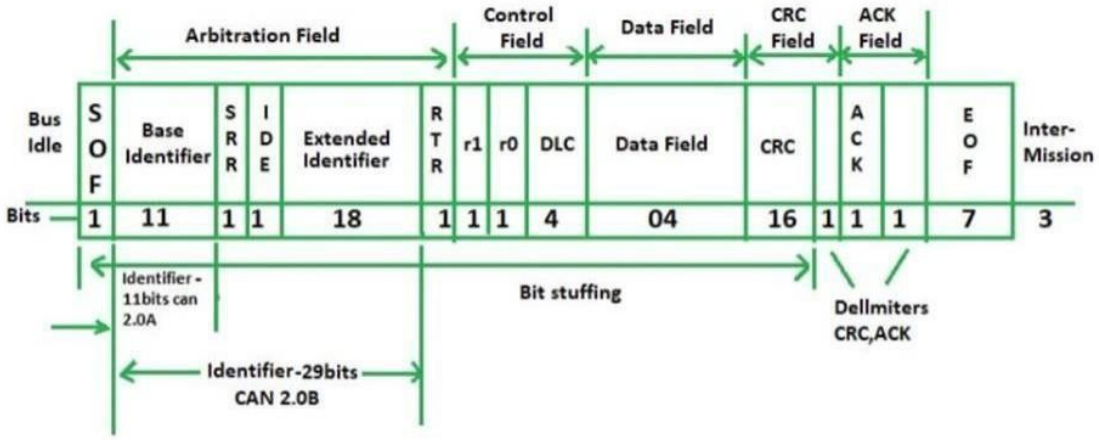
Figure 5.2

- **SOF (start-of-frame) bit** – indicates the beginning of a message with a dominant (logic 0) bit.
- **Arbitration ID** – identifies the message and indicates the message's priority. Frames come in two formats -- standard, which uses an 11-bit arbitration ID, and extended, which uses a 29-bit arbitration ID.

- **RTR (remote transmission request) bit** – serves to differentiate a remote frame from a data frame. A dominant (logic 0) RTR bit indicates a data frame.

  A recessive (logic 1) RTR bit indicates a remote frame.
- **IDE (identifier extension) bit** – allows differentiation between standard and extended frames.
- **R0** – Reversed bit. Not used currently and kept for future use.
- **DLC (data length code)** – indicates the number of bytes the data field contains.
- **Data Field** – contains 0 to 8 bytes of data.
- **CRC (cyclic redundancy check)** – contains a 15-bit cyclic redundancy check code and a recessive delimiter bit. The CRC field is used for error detection.
- **ACK (Acknowledgement) slot** – It comprises the ACK slot and the ACK delimiter. When the data is received correctly the recessive bit in the ACK slot is overwritten as a dominant bit by the receiver.
- **EOF (End of Frame)** – the 7-bit field marks the end of a CAN frame (message) and disables.

2. Extended frame:



29-bit CAN Frame Format

figure 5.3

It is the same as an 11-bit identifier with some added bits.

SRR (Substitute Reverse Request) – The SRR bit is always transmitted as a recessive bit to ensure that, in the case of arbitration between a Standard Data Frame and an Extended Data Frame, the Standard Data Frame will always have priority if both messages have the same base (11 bit) identifier.

R1– It is another bit not used currently and kept for future use.

5.5 Message frame

Four different frames can be used on the bus.

- **Data frames:** These are most commonly used frames and used when a node transmits information to any or all other nodes in the system. Data Frames consist of fields that provide additional information about the message as defined by the CAN specification. Embedded in the Data Frames are Arbitration Fields, Control Fields, Data Fields, CRC Fields, a 2-bit Acknowledge Field, and an End of Frame.

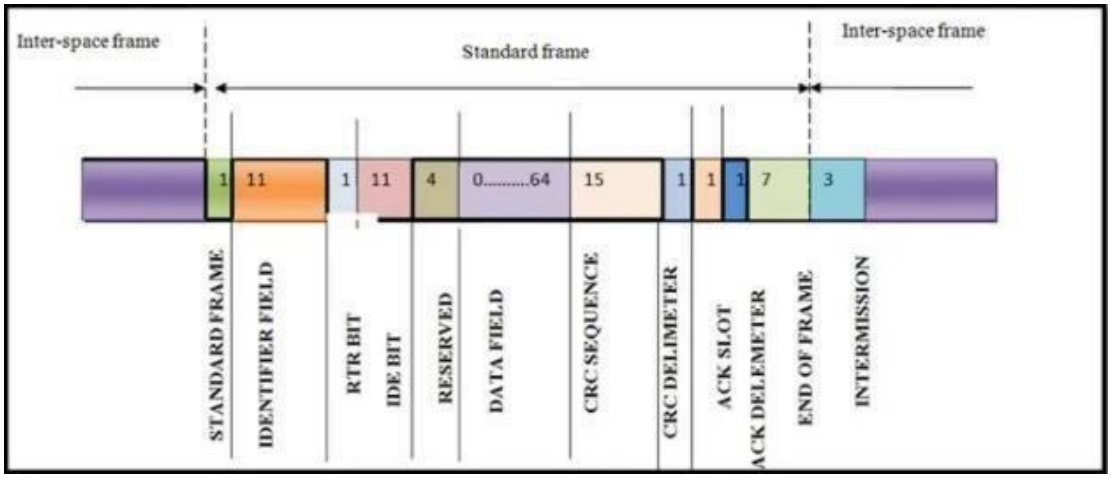


figure 5.4

- **Extended frame:** The purpose of the remote frame is to seek permission for the transmission of data from another node. This is similar to the data frame without the data field and the RTR bit is recessive.
- **Error frames:** If the transmitting or receiving node detects an error, it will immediately abort transmission and send an error frame consisting of an error flag made up of six dominant bits and an error flag delimiter made up of eight recessive bits. The CAN controller ensures that a node cannot tie up a bus by repeatedly transmitting the error frame.

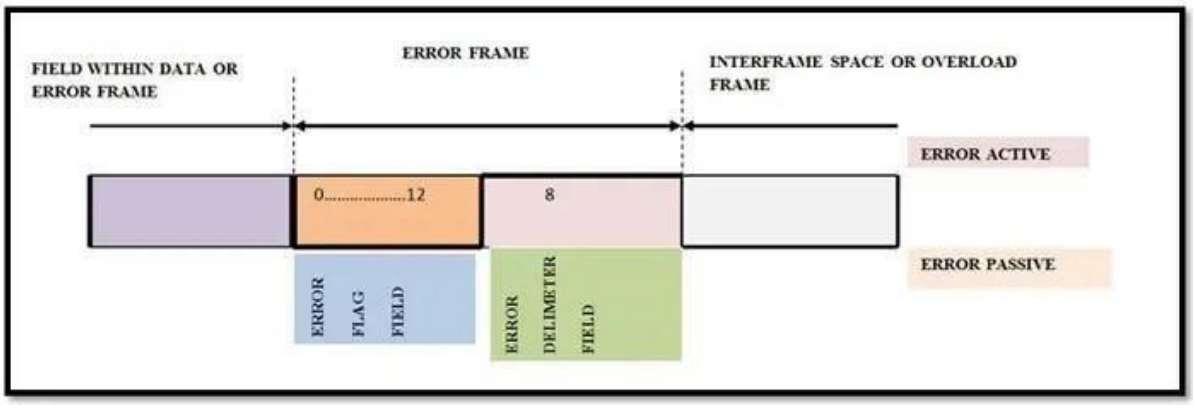


figure 5.5

- **Overload frame:** It is similar to the error frame but used for providing an extra delay between the messages. An Overload frame is generated by a node when it becomes too busy and is not ready to receive.

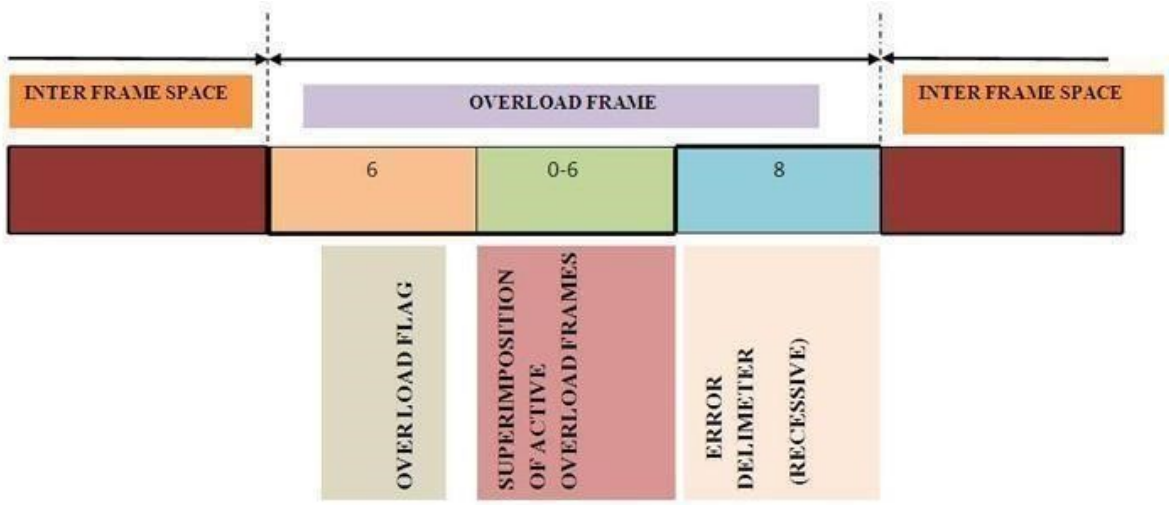


figure 5.6

5.6 CAN Protocol Working Principle

To function properly within a CAN network, each node must consist of three main components that work together to transmit and receive data over the bus:

1. **Microcontroller (Host):** The microcontroller is the central processing unit of the node. It determines:
    a. What data to send based on the application logic
    b. How to process received messages
    c. Whether to respond, log, or take action based on the incoming data

In our project, we use the STM32F103C8T6 (Blue Pill) microcontroller, which includes an integrated CAN controller, allowing efficient interaction with the CAN bus.

2. **CAN Controller:** The CAN controller manages all the low-level operations the CAN protocol. These include:
    - Formatting outgoing data into CAN message frames
    - Managing arbitration and message priorities
    - Generating cyclic redundancy check (CRC) fields
    - Handling error detection and frame acknowledgment
    - Supporting different frame types:
        - **Data Frame** – Carries actual payload
        - **Remote Frame** – Requests data from other nodes
        - **Error Frame** – Indicates protocol violations or hardware faults
        - **Overload Frame** – Adds delay to manage buffer overflow
        - **Inter-Frame Space** – Enforces spacing between consecutive frames In modern microcontrollers like the STM32 Blue Pill, this CAN controller is

    embedded, which simplifies the system design and reduces hardware requirements.
3. **CAN Transceiver:** The CAN transceiver serves as the electrical interface between the CAN controller and the physical CAN bus (wires).

    It performs two essential functions:

    - Transmit: Converts the digital output from the CAN controller into differential voltage signals on the CAN_H (high) and CAN_L (low) lines.
    - Receive: Converts differential voltages from the bus back into logic-level readable by the CAN controller.

Voltage Logic and Signal Levels:

The CAN protocol uses differential signaling to increase noise immunity. Signal levels are interpreted as follows:

- A dominant bit (logical 0) is actively driven by the transceiver by creating a voltage difference between CAN_H and CAN_L.

- A recessive bit (logical 1) is represented by both CAN_H and CAN_L floating to approximately VCC/2 (around 2.5V), and is passively returned by the termination resistors at both ends of the bus.

In idle state (no communication), the CAN bus remains in the recessive state.

It's important to note that due to diode drops and internal limitations of the transceiver, exact voltages (e.g., 0V or 5V) may not be reached. Therefore, CAN_H and CAN_L voltages in dominant mode typically range between 1.5V–3.5V, depending on load and bus configuration.

Termination Resistors:
To ensure signal integrity and minimize reflections on the bus, a 120Ω resistor is placed between CAN_H and CAN_L at each end of the physical bus. These resistors also play a role in maintaining the recessive state during idle periods

5.7 Operation of the CAN Network

In a CAN network, each node (Electronic Control Unit or ECU) does not communicate using physical addresses like traditional communication protocols. Instead, messages are identified using unique identifiers (IDs), which also serve as priority indicators during arbitration.

Although ECUs may have physical identifiers in the system's design documentation, CAN communication is inherently message-based, not address-based. This means that any node can transmit a message, and all other nodes on the bus will receive it, then filter or ignore it based on the message ID.

Arbitration Mechanism: One of the most critical features of CAN is its ability to support multi-master communication

through an efficient bitwise arbitration mechanism. Here's how it works:

1.All nodes can attempt to transmit simultaneously if they have data ready to send.

2.Each transmitting node sends its message ID, starting with the most significant.bit (MSB), while monitoring the bus state at the same time.

3.CAN uses dominant and recessive logic:

a. Dominant (logical 0) wins over recessive (logical 1)

b. If a node sends a recessive bit but reads a dominant bit, it realizes another node has a higher priority and withdraws from arbitration.

4. The node with the lowest numerical ID (i.e., highest priority) continues transmission, while the others wait.

This process ensures that no data collision occurs, and the highest-priority message is always delivered first, preserving determinism and real-time performance.

Example:

Suppose three nodes attempt to transmit messages with IDs:

- Node A: 0x110
- Node B: 0x100
- Node C: 0x090

The arbitration process will detect that Node C (lowest ID = highest priority) wins the bus. Nodes A and B will stop transmitting and retry later.

Error Handling Mechanism:

The CAN protocol includes a sophisticated error detection and confinement system. Both transmitters and receivers constantly monitor the bus for faults. If any kind of inconsistency or violation of the protocol is detected, the node immediately interrupts its transmission and sends an Error Frame.

5.8Interface between Nodes and Communication Network

The system consists of four nodes: three STM32F103C8T6 (Blue Pill) microcontrollers and one Raspberry Pi 4, which acts as the main controller. All nodes are connected via a common CAN (Controller Area Network) bus.Each STM32 node is equipped with a built-in CAN controller, while the Raspberry Pi 4 interfaces with the CAN bus through an external transceiver module.

To ensure proper communication, each node is assigned a unique CAN message ID. This ID is embedded in every message sent across the network. While all

messages travel through the shared CAN bus, only the node with the matching ID will receive and process a particular message. This is enabled by the CAN hardware filters and acceptance masks built into each STM32 CAN unit.

The Raspberry Pi 4 serves as the master node, responsible for:

- Coordinating system behavior
- Sending commands to STM32 nodes
- Receiving status or sensor data from STM32s

There are techniques in the CAN unit itself that can handle transmitting and receiving messages.

a)CAN Filtering:

The filtering is done by the arbitration identifier of the CAN frame. This technique is also used when monitoring a CAN bus, to focus messages of importance using an identifier and mask. These allow a range of IDs to be accepted with a single filter rule. When a CAN frame is received, the mask is applied. This determines which bits of the identifier will be used to determine if the frame matches the filter.

b)CAN FIFO Buffer:

First in First Out concept using CAN to Store multiple Messages Received in RAM as Receiving each message Individually keeps interrupting the CPU for every single Message received. It will make the system overhead especially in multitasking systems.

5.9 CAN in Action

In our system, the Controller Area Network (CAN) is actively used to transmit and receive structured messages between intelligent nodes. One of the key design choices in our

implementation is to represent the message data payload in JSON format. JSON (JavaScript Object Notation) is a lightweight, text-based data format that uses key-value pairs, making it readable and easy to parse on platforms like the Raspberry Pi 4, which acts as the central controller and logging interface in our architecture.

a)Transmitting a CAN message:

Transmitting a message typically requires loading the identifier, RTR, ID, Data length, and the Data into Transmit structure.

a)Receiving a CAN message:

Receiving a message typically requires loading filtering id to set filter structure and monitoring the FMP bit with interrupt or polling to notify user application when a CAN message has been received.

This mechanism allows the Raspberry Pi 4 (the main controller) to send control messages and receive status updates from the STM32 Blue Pill nodes in a structured and human-readable format.

CHAPTER 6 :
DBus

6.1 Introduction to dbus

What is D-bus?

DBus is an IPC system that allows multiple applications to communicate on the same machine. It supports both system-wide and per-user communication channels.

Dbus-architecture:

A-System Bus: For communication between system services (e.g.,hardware-related daemons like NetworkManager).

B-**Session Bus**: For communication between user applications in a graphical or shell session.

core concepts summary:

- **Bus** : A communication channel (System or Session Bus).
- **Service Name:** Unique identifier for a DBus application (e.g., `com.example.MyService`). **Object Path:** Path within the service to expose functionalities (e.g., `/com/example/MyService`).

- **Interface:** Logical grouping of methods, signals, and properties (e.g., `com.example.MyService.Interface`).
- **Method:** Callable function exposed via DBus.
- **Signal:** Event notification emitted by a service.
- **Property:** Readable and/or writable values exposed by the object.

6.2 DBUS Concepts Service Object Interface service:

A service is the DBus application itself. It registers a unique:

- **Service Name:** (like `com.example.MyService`) on the bus. Only one process can own a service name at a time.
- **Object:**
  - Each service can register one or more objects using an object path (e.g., `/com/example/MyService`).
  - Objects expose methods, signals, and properties. object paths are logical identifiers within the context of the D-Bus messaging system. They are defined by developers, and their structure is meant to organize and locate D-Bus-exposed objects, not files.
- **Interface:**
  - An interface defines the functionality (methods, signals, and properties) that an object provides. Interfaces group related behavior and are identified by dotted names (e.g., `com.example.MyService`).

6.3 Session Bus Vs System Bus:

Feature	**Session Bus**	**System Bus**

Scope	Per user session	System-wide
Access	Apps running in a user session	Services like `NetworkManager`, `udev`
Usage Example	GUI apps communicating with each other	Background daemons exposed to all users
Command	`QDBusConnection::sessionBus()`	`QDBusConnection::systemBus()`
Debug Tool	`dbus-monitor --session`	`dbus-monitor --system`

6.4 D-Bus Applications

There are many, many technologies in the world that have "Inter-process communication" or "networking" in their stated purpose: CORBA, DCE, DCOM, DCOP, XML-RPC, SOAP, MBUS, Internet Communications Engine (ICE), and probably hundreds more. Each of these is tailored for particular kinds of application. D-Bus is designed for two specific cases:

- Communication between desktop applications in the same desktop session; to allow integration of the desktop session as a whole, and address issues of process lifecycle (when do desktop components start and stop running).
- Communication between the desktop session and the operating system, where the operating system would typically include the kernel and any system daemons or processes.

For the within-desktop-session use case, the GNOME and KDE desktops have significant previous experience with different IPC solutions such as CORBA and DCOP. D-Bus is built on that experience and carefully tailored to meet the needs of these desktop projects in particular. D-Bus may or may not be appropriate for other applications; the FAQ has some comparisons to other IPC systems.

The problem solved by the systemwide or communication-with-the-OS case is explained well by the following text from the Linux Hotplug project:

> A gap in current Linux support is that policies with any sort of dynamic "interact with user" component aren't currently supported. For example, that's often needed the first time a network adapter or printer is connected, and to determine appropriate places to mount disk drives. It would seem that such actions could be supported for any case where a responsible human can be identified: single user workstations, or any system which is remotely administered.
>
> This is a classic "remote sysadmin" problem, where in this case hotplugging needs to deliver an event from one security domain (operating system kernel, in this case) to another (desktop for logged-in user, or remote sysadmin). Any effective response must go the other way: the remote domain taking some action that lets the kernel expose the desired device capabilities. (The action can often be taken asynchronously, for example letting new hardware be idle

until a meeting finishes.) At this writing, Linux doesn't have widely adopted solutions to such problems.
However, the new D-Bus work may begin to solve that problem.

D-Bus may happen to be useful for purposes other than the one it was designed for. Its general properties that distinguish it from other forms of IPC are:

- Binary protocol designed to be used asynchronously (similar in spirit to the X Window System protocol).
- Stateful, reliable connections held open over time.
- The message bus is a daemon, not a "swarm" or distributed architecture.
- Many implementation and deployment issues are specified rather than left ambiguous/configurable/pluggable.
- Semantics are similar to the existing DCOP system, allowing KDE to adopt it more easily.
- Security features to support the systemwide mode of the message bus.

6.4.1 Native Objects and Object Paths

Your programming framework probably defines what an "object" is like; usually with a base class. For example: java.lang.Object, GObject, QObject, python's base Object, or whatever. Let's call this a *native object*.

The low-level D-Bus protocol, and corresponding libdbus API, does not care about native objects. However, it provides a concept called an *object path*. The idea of an object path is that higher-level bindings can name native object instances, and allow remote applications to refer to them.

The object path looks like a filesystem path, for example an object could be named /org/kde/kspread/sheets/3/cells/4/5. Human-readable paths are nice, but you are free to create an object named /com/mycompany/c5yo817y0c1y1c5b if it makes sense for your application.

Namespacing object paths is smart, by starting them with the components of a domain name you own (e.g. /org/kde). This keeps different code modules in the same process from stepping on one another's toes.

6.4.2 Methods and Signals

Each object has *members*; the two kinds of members are *methods* and *signals*. Methods are operations that can be invoked on an object, with optional input (aka arguments or "in parameters") and output (aka return values or "out parameters"). Signals are broadcast from the object to any interested observers of the object; signals may contain a data payload.

Both methods and signals are referred to by name, such as "Frobate" or "OnClicked".

6.4.3 Interfaces

Each object supports one or more *interfaces*. Think of an interface as a named group of methods and signals, just as it is in GLib or Qt or Java. Interfaces define the *type* of an object instance.

DBus identifies interfaces with a simple namespaced string, something like org.freedesktop.Introspectable. Most bindings will map these interface names directly to the appropriate programming language construct, for example to Java interfaces or C++ pure virtual classes.

6.5 D-Bus Architecture

Here, you'll discuss how D-Bus works at an architectural level.

Core Components:

Bus Daemon (D-Bus Daemon): The central server that manages all messages between clients. Explain its role as an intermediary ensuring messages reach the correct destination.

Buses (Bus Types):

- System Bus: For system-wide communications between services and privileged processes (e.g., NetworkManager, UDisks).
- Session Bus: For communications between user applications within a single session (e.g., desktop applications, file browsers).

Messages:

- Method Calls: To request the execution of a specific function in another process and receive a reply (Request/Reply).
- Signals: To send Event Notifications in a one-to-many manner (e.g., Broadcast).
- Error Messages: To report errors.
- Reply Messages: Responses to Method Calls.

Core D-Bus Concepts:

- Bus Names: Unique names for processes on the bus (e.g., org.freedesktop.NetworkManager).
- Object Paths: Identify objects within a specific process (e.g., /org/freedesktop/NetworkManager/Settings).
- Interfaces: Define the Methods, Signals, and Properties that an object provides (e.g., org.freedesktop.NetworkManager.Settings.Connection).
- Members: The Methods and Signals.
- Serialization and Deserialization: How data is converted into a format that can be transferred across the bus and then re-converted.

6.6 D-Bus Use Cases

Present practical examples to illustrate the importance of D-Bus.

1.In Desktop Environments:Device Management: Connecting and disconnecting USB devices (UDisks).

- Network Management: Controlling Wi-Fi and connections (NetworkManager).
- Audio: Controlling volume levels (PulseAudio/PipeWire).
- Power Management: Shutting down, restarting, suspending. Notifications: Sending system notifications to applications.

2.In Embedded Systems:

- Communication between different system components (e.g., between a user interface and a hardware control server).
- Automotive Linux applications.
- Smart Home Systems.
- Power and resource management in resource-constrained devices.
- D-Bus's Role with systemd: Explain how systemd heavily relies on D-Bus for managing services and processes.

6.6.1 D-Bus Advantages and Disadvantages

Discussing the strengths and weaknesses of D-Bus.

Advantages:

- Simplicity and Standardization: Provides a unified and simple API for communication.
- 2.Language Agnostic: Can be used with various programming languages (C, C++, Python, Java, Rust, etc.).
- Plug-and-Play: Facilitates dynamic discovery and communication with available services.
- Event-Driven: Supports sending signals, which simplifies building event-driven systems.
- Security: Offers simple mechanisms for access control and permissions (Policy-based security).
- 1.Robustness: A design that reduces single points of failure.

Disadvantages:

- Performance: May not be the optimal choice for communicating very large amounts of data or applications requiring very low latency (e.g., real-time video).
- Initial Complexity: Understanding the initial concepts (Objects, Interfaces, Paths) can be challenging for beginners.
- Limited Remote Usage: Primarily designed for local communication on the same machine, not for network communication (though solutions exist, it's not its primary use case).
- Memory Consumption: In some resource-constrained environments, memory consumption might be an issue.

6.6.2 Performance

Considerations: Discuss D-Bus performance:
Performance Measurement: How to measure D-Bus performance in a specific application.Factors Affecting Performance:

- Number of messages sent.
- Size of data in messages.
- Number of Context Switches.
- Performance Optimization:
- Reduce the number of D-Bus calls.

Aggregate data into larger messages instead of sending multiple small ones. Use Asynchronous methods where possible.

6.6.3 Comparison with other IPC Mechanisms

Put D-Bus in a broader context by comparing it with other IPC mechanisms.

Sockets: Advantages and disadvantages of D-Bus compared to Sockets (D-Bus builds on Unix Domain Sockets).

1. Pipes: When D-Bus is better than Pipes.
2. Shared Memory: Differences in usage and complexity.
3. RPC Frameworks: How D-Bus differs.
4. Mention when to choose D-Bus and when another option is better.

6.7 System D-Bus

6.7.1 System D-Bus: The Core Basics

The System D-Bus is a single, unified message bus for the entire system. It acts as a communication hub for all processes and services that require system-level communication, regardless of which user started them or which session they belong to.

Primary Goal: To enable privileged components and services (like device drivers, network managers, and logging services) to communicate with each other and with other applications in a secure and organized manner.

Service Operation: The System D-Bus is run by the dbus-daemon process, which typically operates with root privileges or a special system user (like dbus). This gives it the ability to manage system-level communications.

Single Point of Connection: All services and applications that wish to communicate via the System D-Bus connect to the same dbus-daemon instance, which manages message routing.

6.7.2 Security Policies (Policy Files) in System D-Bus

Given the sensitivity of System D-Bus and its role in managing system resources, security is paramount. Permissions on System D-Bus are precisely controlled using Policy Files. Location: These files are typically found in /etc/dbus-1/system.d/ or /usr/share/dbus1/system.d/.

XML Format: Policy files are written in XML format and define who has the right to:

Own a service name: Who is allowed to register a specific name on the bus (e.g., org.freedesktop.NetworkManager).

Send messages to a specific service: Who is allowed to invoke methods or send signals to a particular service.

Receive messages from a specific source: Who is allowed to receive signals or replies from a particular service.

Message types: Specify allowed message types (method_call, signal, error, method_return).

Paths, Interfaces, and Members: Permissions can be specified at a more granular level, such as allowing access to only a specific method within a certain interface.

Policy Rules: Rules are applied based on:

1. User: Specify permissions for a particular user.

2. Group: Specify permissions for a group of users.
3. Default: Rules applied to all other users or processes.

6.7.3 Key Functions of System D-Bus

System D-Bus is used in many critical operating system functions:

- Hardware Management: When you connect a USB device, udev (which handles device detection) sends a signal over System D-Bus to notify other services and applications about the new device.

- Network Management: The NetworkManager service uses System D-Bus to communicate with other system components, such as network settings, or to inform applications about connection status changes.

- Power Management: Power management services send signals via System D-Bus to alert applications that the battery is low or that the system is about to enter sleep mode.

- User and Session Management: systemd-logind uses System D-Bus to track user sessions, manage login/logout processes, and control shutdown and reboot operations.

- System Background Services (Daemons): Many services running in the background use System D-Bus to expose their Application Programming Interfaces (APIs) to other applications or to receive commands from administrative tools.

- System-wide Notifications: Services can send general system-wide notifications to be displayed to users (e.g., system updates, hardware issues).

6.7.4 Security and Considerations:

Security is at the heart of System D-Bus. Any vulnerability in it could affect the stability and security of the entire system.

Principle of Least Privilege: This principle is strictly applied. Policy files should grant the minimum necessary permissions for services and applications to perform their functions.

Authentication and Authorization:

- D-Bus uses simple mechanisms for authentication (like User ID) and authorization (via policy files).
- On Linux, D-Bus often uses socket options like SO_PEERCRED to verify the identity of the connecting process.

- Polkit (PolicyKit): Often, D-Bus is integrated with Polkit for an additional layer of permission control. Polkit allows for defining more complex policies that involve user interaction (e.g., requiring a password to authenticate a specific action).

- Monitoring: Tools like dbus-monitor --system can be used to monitor activity on the System D-Bus and detect any unexpected behavior or unauthorized access attempts.[25]

6.7.5 Programmatic Interaction with System D-Bus:

When programming and interacting with System D-Bus, you typically use high-level bindings that provide a simpler interface.

Libraries such as:

1. GDBus (for C with GLib): Widely used in GNOME environments and system applications.
2. QtDBus (for C++ with Qt): Used in KDE environments and Qt applications. sd-bus (part of systemd): A modern, low-level C interface providing good performance.

3.Python-dbus:A Python binding that allows easy interaction with D-Bus.

General Steps for Interaction:

1.Connect to the bus: Establish a connection to the System D-Bus.
Own a name (Optional for clients): If your service will provide functionality, it must request to own a unique name on the bus.

2.Create an object: Define the object that will represent your service on the bus.

3.Define interfaces, methods, and signals: Specify the methods and signals your service will provide.

4.Send/Receive messages: Use the library's APIs to send method calls,receive method returns, send signals, or receive signals.

6.7.6 System D-Bus using QT C++

QtDBus simplifies interacting with the System D-Bus in C++, allowing your applications to easily call functions on system services like NetworkManager and receive signals from them via the QDBusConnection::systemBus() object.

To offer your own service on the System Bus, you can use qdbusxml2cpp to generate adaptor code, then register your service with registerService() and registerObject().

Always remember that System D-Bus security requires specific policy files to grant the necessary permissions.

6.8 Challenges and Future Directions

Conclude by discussing current challenges and future prospects for D-Bus.

Challenges:

- More extensive handling of Distributed Systems.
- Improving performance in specific scenarios.
- Simplifying the development process for new developers.

Future Directions:

Evolution of the protocol itself or the emergence of alternatives/complements.

Its role in new operating systems and emerging technologies.

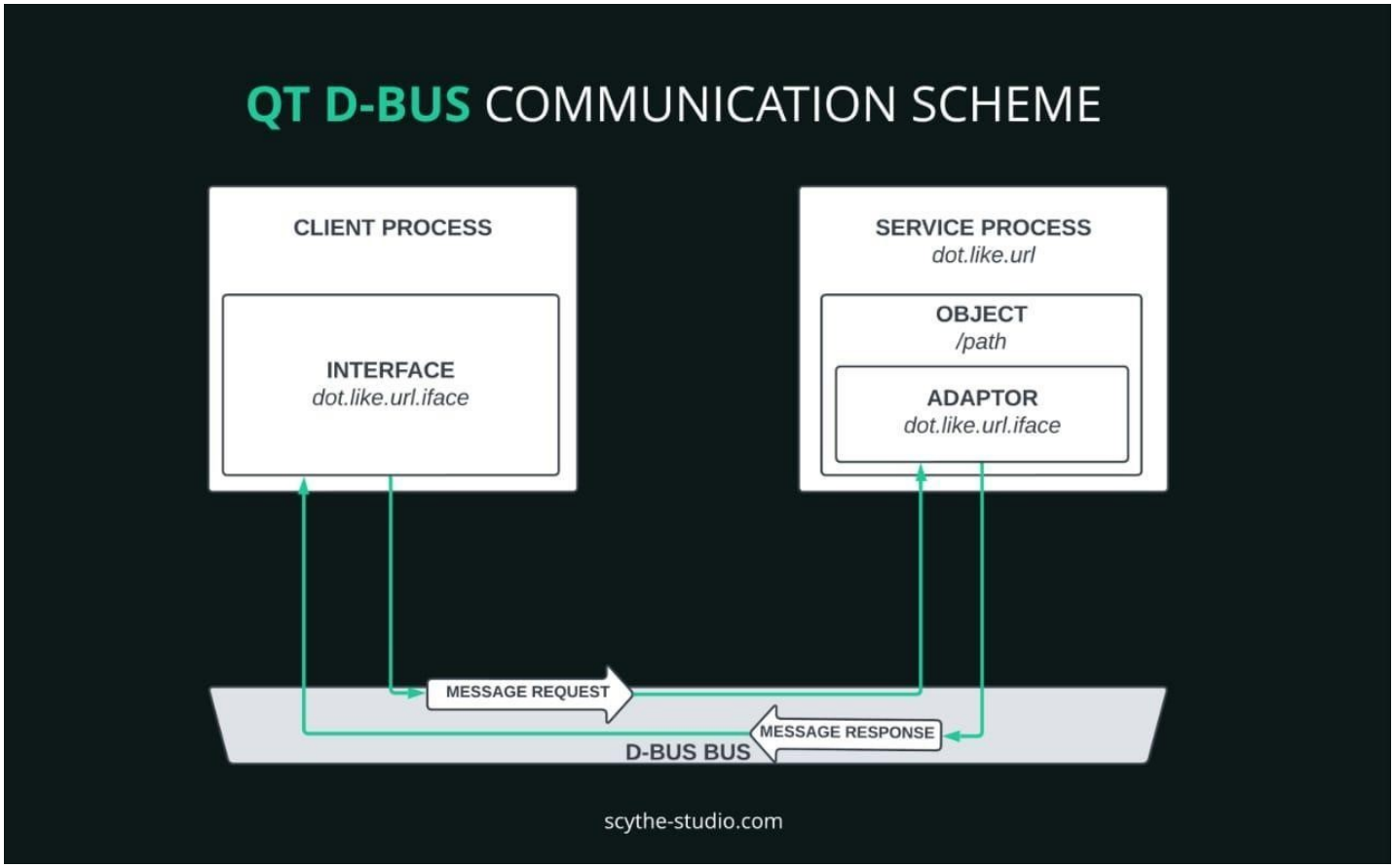

QT D-BUS COMMUNICATION SCHEME
CLIENT PROCESS
INTERFACE
dot.like.url.iface
SERVICE PROCESS
dot.like.url
OBJECT
/path
ADAPTOR
dot.like.url.iface
MESSAGE REQUEST
MESSAGE RESPONSE
D-BUS BUS
scythe-studio.com

figure 6.1

Module C

Subsystems

Chapter 7 : Vision Assistant System.

Chapter 8 : Lane Departure Warning (LDW).

chapter 9 : Firmware Over The Air (FOTA).

chapter 10 : logger (DTC Handler).

CHAPTER 7:
Vision Assistance System

7.1 Introduction

In modern intelligent transportation systems, vision-based safety enhancements have become vital, especially in conditions of poor visibility. Our project focuses on developing a real-time driver assistance system using computer vision, artificial intelligence, and high efficiency hardware to alert the driver to road hazards.
Initially designed for night vision, the system was later enhanced to work in any visibility compromised condition, such as fog, dust, or rain.

7.2 Problem Statement

At night, there is 60 percent less traffic on the roads. Despite this, 50 percent of fatal accidents happen after dark, as our ability to perceive distance and detect objects is severely impaired at night.

Now with the combined power of IR Cameras and AI, night vision has become more accurate and safer, Reducing the risk of accidents and enhancing driving safety in the dark.

The Night Vision System is designed to provide enhanced visibility in low-light or completely dark conditions. The system captures images using thermal cameras or infrared cameras and uses AI to obtain distance and object data. These images are processed using computer vision and artificial intelligence techniques to improve clarity and detect objects. The enhanced images are then displayed on a screen for the driver.

7.3 System Objectives

The primary goals of the Vision System are:

- ➢To assist drivers during night-time or low-visibility conditions by detecting and classifying obstacles ahead.
- ➢To provide real-time visual analysis using an IR camera and object detection models.
- ➢To estimate the distance and motion state (moving or static) of detected objects.
- ➢To warn the driver in case of potential danger, enabling timely responses.
- ➢To offer a high efficiency and portable solution that can be deployed on a Raspberry Pi 4.

7.4 Working Methodology

The vision system works in continuous cycles:

1. **Image Acquisition**: IR camera captures real-time frames.
2. **Preprocessing:** Frames are resized, normalized, and converted to the correct color format for YOLOv4-Tiny.
3. **Object Detection:** YOLOv4-Tiny identifies common objects like people, vehicles, animals, etc.
4. **4. Motion Status Detection:** By comparing object positions in consecutive frames, the system estimates if the object is moving or static.
5. **Distance Estimation:** The size and location of bounding boxes help estimate object distance.
6. **Alert Generation:** If a hazard is detected within a critical distance, an alert is issued.

7.5 System Requirements

The system is composed of both hardware and software components:

Software:

- ➢**Operating System:** YOCTO OS.
- ➢**Programming Language:** Python.
- ➢**Libraries Used:** OpenCV for image handling and YOLOv4-Tiny for object detection.

Hardware:

- ➢**IR (Infrared) Camera:** Captures the road in dark or conditions.
- ➢**Raspberry Pi 4:** Serves as the processing unit. Receives camera frames, runs object detection, and provides alerts.
- ➢**Power Source:** Typically a car battery.
- ➢**Warning Interface:** (e.g., Buzzer, LED, or audio output) alerts the driver.

7.6 Why?

7.6.1 Why Use Artificial Intelligence (AI)?

Traditional image processing techniques (like edge detection, contour analysis) are not reliable in poor lighting or cluttered environments. AI-based approaches offer:

- ➢**High accuracy** in detecting multiple object types.
- ➢**Robust performance** under different lighting/weather conditions.
- ➢**Ability to generalize** across different environments.
- ➢**Adaptability:** The model can be retrained with new data to support additional object classes. So you can say that AI brings intelligence into the system, allowing it to "understand" the road scene like a human would.

7.6.2 Why YOLOv4-Tiny?

YOLO stands for "You Only Look Once," a family of real-time object detection algorithms.

Reasons for using YOLOv4-Tiny:

- ➢**Lightweight:** Optimized for devices with limited processing power.
- ➢**Fast Inference:** Runs efficiently on the Raspberry Pi.
- ➢**High Accuracy:** Maintains strong accuracy while being smaller than full YOLO models.
- ➢**Real-Time Capable:** Delivers acceptable frame rates on embedded systems.
- ➢**Well-supported:** Available in many frameworks (Darknet, TensorFlow, PyTorch, etc.).

In short, YOLOv4-Tiny gives us the best trade-off between speed and accuracy. [

7.6.3 Why Raspberry Pi and Not STM32?

STM32 microcontrollers are excellent for low-power and real-time control systems, but not for tasks like object detection.

Reasons for choosing Raspberry Pi 4:

- ➢**Runs full Linux OS:** Can install Python, OpenCV, deep learning libraries.
- ➢**Supports USB and Camera Modules:** Easily interfaces with IR camera.
- ➢**4-core CPU and 4/8 GB RAM**: Enough for running YOLOv4-Tiny.
- ➢**GPU Acceleration (with effort)**: Supports OpenCL/Vulkan via GPU.
- ➢**HDMI Output:** Can display visual feedback to the driver.
- ➢**Affordable and Portable**

STM32 would require additional processing modules (e.g., NPU, external co-processors), increasing complexity and cost.

7.7 Challenges

While developing the system, we encountered several issues:

➢Heat on Raspberry Pi during processing → Solved using a fan.

➢Open camera on Raspberry Pi → Solved using Picamera library.

➢Low frame rate with full-size YOLOv4 → Switched to Tiny version.

➢Distance estimation without a depth camera → Used pixel size heuristics.

7.8 System Extensions and Capabilities

Beyond its core functionality, our system already supports multiple advanced capabilities that are typically found in commercial automotive solutions:

➢ADAS Integration: The vision system is part of a broader Advanced Driver Assistance System (ADAS), where it communicates with other subsystems (e.g., lane keeping system in our project using the main system). Detected hazards can influence driving decisions or be logged for further action.

➢ FOTA (Firmware Over-the-Air): The system can receive software or model updates remotely. This allows ongoing improvements in object detection performance and system logic without the need for physical access to the hardware.

7.9 System Block Diagram

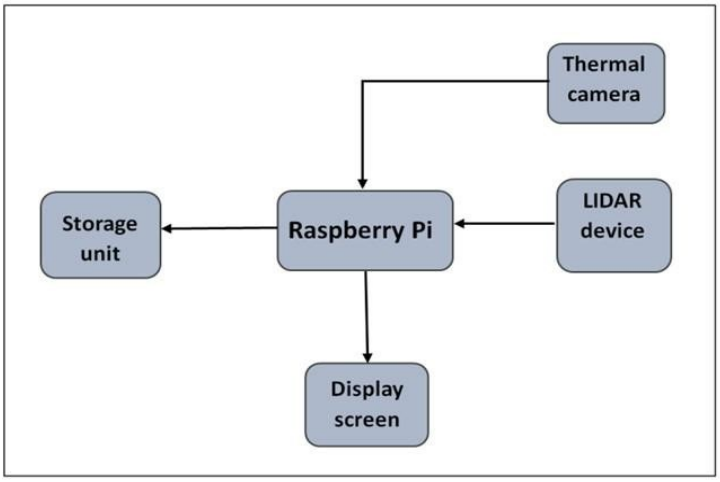


figure 7.1

7.10 System Flowchart

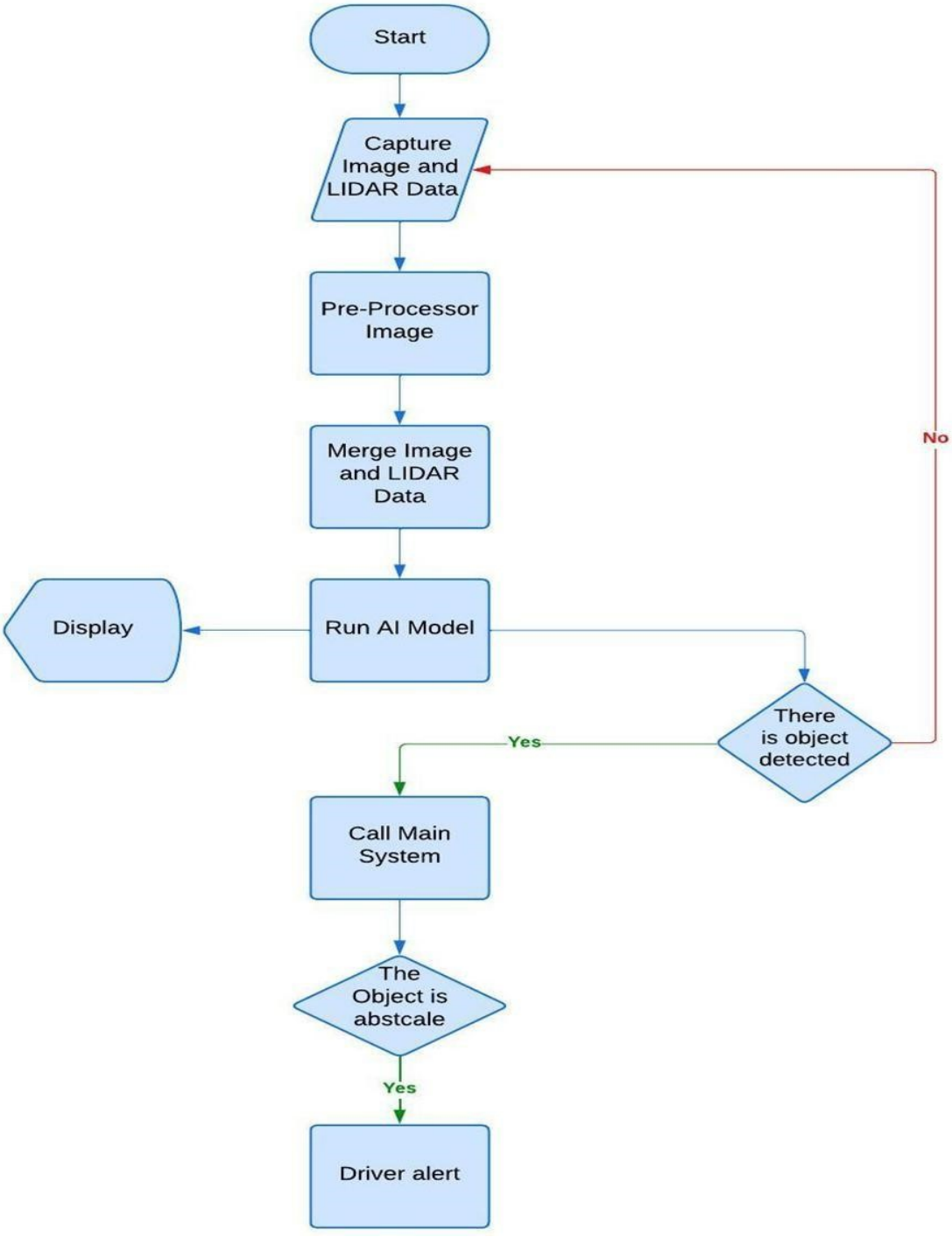


figure 7.2

CHAPTER 8:
LDW (Lane Departure Warning)

8.1 Introduction

Micro-controllers are used to develop intelligent systems that can be found in different applications in our daily lives. In this project, we aim to design lane departure warning (LDW) using stm32f103C8T6 "Blue-pill".LDW we are developing autonomous cars using the IR sensor or camera. In this car, we used two IR sensors to find the lines. If white line detected means the car will turn in another direction.

The main motivation for building this system is LDW works. This system makes the car move in the same line without crossing it and it reduces many of the problems that people suffer from.

The goal of this system is to:

1. Helping drivers who do not have high driving skills.
2. Helping drivers with special needs.
3. Reducing traffic accidents

8.2 Problem Statement

Modern road traffic conditions have seen a significant rise in accidents due to lane departures caused by driver inattention or fatigue. Unintentional drifting out of a lane is a critical safety concern, especially at high speeds on highways. Drivers often lose focus, which may lead to the vehicle moving off-course, potentially causing collisions or running off the road. Current manual driving systems rely entirely on human input, with no safeguards against such errors. As autonomous and semi-autonomous driving technologies advance, there is a growing need to integrate intelligent systems that can monitor lane positions and take corrective actions automatically. The problem arises

in accurately detecting lane positions and adjusting the vehicle's trajectory without human intervention.

Therefore, a reliable (LDW) system is needed. This system must continuously monitor the

vehicle’s position in the lane using sensors and perform necessary steering adjustments. The challenge also includes minimizing false detections and ensuring smooth movement while enhancing road safety.

8.3 System Description

The proposed LDW system is designed to help vehicles stay within lane boundaries using an array of infrared (IR) sensors. As shown in the flowchart, the system starts by constantly reading inputs from IR sensors positioned on the underside or front of the vehicle. These sensors detect the presence or absence of lane markings.

The logic control unit (e.g., a microcontroller) interprets these readings to determine the vehicle's position relative to the lane. If the lane is not properly detected, the system decides the required correction direction: turn left if the vehicle is veering right, or turn right if it's moving toward the left lane. If both sensors detect the lane correctly (meaning the vehicle is centered), the system instructs the car to continue moving straight. These actions are performed repeatedly in a loop, creating a feedback mechanism.

The system ensures continuous adjustments for smooth lane alignment, improving safety. The design can be implemented in either fully autonomous or driver-assist systems.

8.4 System Requirements

1. **Sensors**: A set of IR sensors capable of detecting lane markings in various lighting and weather conditions.

2. **Processing Unit**: A microcontroller (such as Arduino or Raspberry Pi) to process sensor data and control vehicle movement.

3. **Actuators**: Steering actuators or servos to perform left or right adjustments based on logic.
4. **Power Supply**: Reliable power source to ensure continuous operation of sensors and processor.

5. **Chassis**: A test vehicle or car model to integrate hardware components.

6. **Software Logic**: A program that includes sensor reading, decision-making, and motor control logic, as shown in the flowchart.

7. **Testing Environment**: Marked track or lane simulation to test and validate system performance.

8. **Safety Mechanisms**: Fail-safes or overrides for manual control in case of system malfunction.

9. **Debugging Tools**: LEDs, serial monitors, or displays to track sensor outputs during development.

8.5 Expected Results

The system is expected to successfully keep the vehicle centered within its designated lane by making continuous adjustments based on IR sensor feedback. When the vehicle starts drifting, it should accurately detect the direction of drift and immediately execute a corrective turn.

If the vehicle is centered, it should move forward without deviation. The correction should be smooth to avoid jerky movements and provide a comfortable driving experience. In test runs, the car should demonstrate high accuracy in detecting lane markings and quickly respond to any lateral deviations.

The response time must be fast enough to prevent crossing lane boundaries, especially at higher speeds. The system should be resilient to minor signal noise and function reliably on common road surfaces.

Overall, the LDW should reduce the chances of lane departure and serve as a foundational component for more advanced driver-assist or autonomous systems. The successful implementation of the system will be indicated by consistent lane following performance under different testing conditions.

8.6 Flowchart

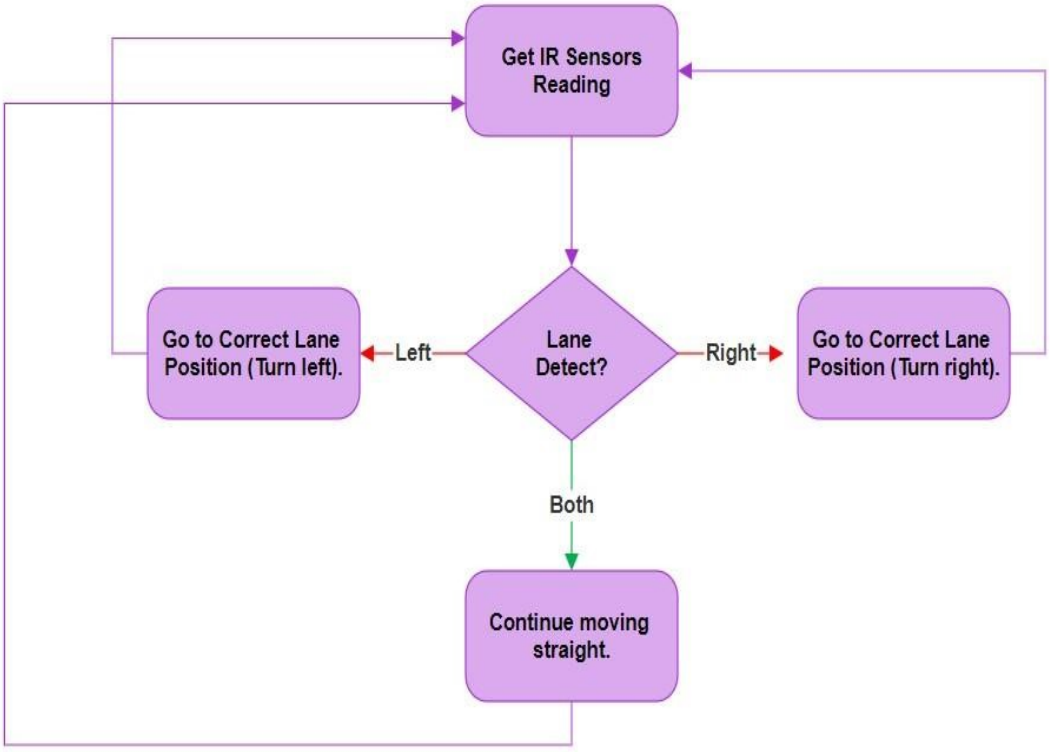


figure 8.1

CHAPTER9:
FOTA

9.1 Introduction

9.1.1 Overview of Firmware Updates

The increasing importance of updates in the world of connected devices (IoT) and Embedded Systems; to maintain security, fix bugs, and add new features. Traditional update methods (e.g., manual update via USB, JTAG/SWD); discussing their challenges such as cost, human effort, and geographical limitations.

The emergence of the need for remote updates to enable efficient maintenance and rapid deployment of improvements.

9.1.2 Concept of FOTA (Firmware Over-The-Air

A comprehensive definition of FOTA: a mechanism for updating device firmware remotely over a wireless connection.

Key benefits of FOTA: reducing operational costs, accelerating the product life cycle, improving security through quick vulnerability patches, and the ability to add continuous value to the product after sale.

Inherent challenges in FOTA implementation: ensuring reliability (what if the update is interrupted?), security (preventing unauthorized updates), and efficient resource utilization (network and power on the device).[47]

9.1.3 Tools and Platforms Used in the Project

Mender Tool: A brief introduction to Mender as an open-source solution for FOTA update management, focusing on its core features such as reliability, security, and device fleet management.

Raspberry Pi: Explanation of why it was chosen as the primary platform for implementing the

"Mender client" side, which runs Linux, due to its flexibility and computational capabilities

STM32: An overview of its use as a resource-constrained embedded device platform, and how it integrates into a Mender-managed FOTA system (either as a subordinate device or through the Raspberry Pi).

9.1.4 Project Objectives

Design and implement a secure and reliable FOTA system using the Mender Tool.

Demonstrate Mender's applicability to different platforms such as Raspberry Pi and STM32. Investigate the security aspects of update processes, including digital signing and data integrity verification.

Evaluate the effectiveness of Mender's automatic rollback mechanisms.

9.1.5 Thesis Outline

An overview of the content of the upcoming chapters to guide the reader.

9.1.6 What is FOTA?

Firmware Over-The-Air (FOTA) is a Mobile Software Management (MSM) technology in which the operating firmware of a device is wirelessly upgraded and updated by its manufacturer.

Firmware runs in the background without any input from the user and is there to ensure the device's hardware runs properly.

The process of downloading these updates wirelessly usually takes little time, depending on the connection speed and the size of the update.

This saves businesses the time and money spent sending a technician to have each one of their cellular devices physically upgraded or updated.

Bugfix allows manufacturers to repair faulty units and remotely update software updates. This method typically involves the consumer downloading and updating mobile device firmware through a manufacturer's website or server.

FOTA updates are generally accessible through the device menu, software, or firmware update. Security updates are constantly being released by the manufacturers, which includes a list of all known vulnerabilities during the Data transfer process. All devices must be updated regularly to avoid potential exploitation.

Cost-Efficient and better-managed firmware update: Most of the vehicle development process is spread across numerous stakeholders and geographical locations.
In such a case, performing any firmware update could be a challenging task that involves multiple revisions and modifications.

Thus, an over-the-air update through a wireless network eases the task while lowering any additional cost and time consumed for multiple software firmware updates during the entire lifecycle of a car.
IHS Automotive has estimated that the total OEM cost savings from the OTA (firmware and software) update process will grow to more than $35 billion in 2022 from $2.7 billion in 2015..
Upgrade the firmware safely, anytime & anywhere: One of the critical uses of a FOTA update is to send an update to the car, post-decommissioning.

FOTA enables the cars to be upgraded remotely, without having the end-user worry about a software-related recall or update.

Many OEMs have relied on over-the-air upgrades to fix major functional faults in their automotive software.

With FOTA, often most of such firmware bugs can be fixed without the need for a vehicle recall by either sending a new firmware or sending a small patch to update the existing firmware with the bug fixing module.

For example, some time back, Tesla's Model S caught fire in a collision. Tesla dealt with this by sending out an update that changed the suspension settings. Since the update, no further fires have been reported, (although it's not very clear if this was due to the update).

Lowers the Time to market: As already discussed above (point 1) FOTA has been helping the OEMs with reduced time-to-market by updating firmware while the vehicle is still on the production line.

9.1.7Examples of FOTA

In a well-known example from 2016, Tesla used FOTA to update each of their cars with the ability to self-park.

Without FOTA, each car would have had to either be recalled or visited by a Tesla technician for these updates to be installed. FOTA is particularly useful when it comes to IoT systems, especially those with large numbers of connected devices that require frequent updates.

For example, updating hundreds of sensors measuring soil moisture levels across a large farm would be a near-impossible task using the traditional method, in which each sensor would have to be retrieved, connected to a computer or handheld device, reprogrammed with the new update, and placed back in the field, creating unnecessary costs and performance disruption.

9.2 Theoretical Foundations of FOTA

9.2.1 Generic FOTA System Architecture

Detailed breakdown of key components: Update Server, Update Client on the device, and Communication Channel.
Explanation of a typical update flow: from update request, through download, to installation and reboot.
Review of common protocols used in FOTA (e.g., HTTP/HTTPS, MQTT, CoAP) and a comparison in terms of suitability and limitations.

9.2.2 Security Challenges in FOTA

- Firmware Authenticity: Ensuring that the update comes from a trusted source.
- Importance of Digital Signatures and Hashing Functions.
- Data Confidentiality: Protecting update data from eavesdropping during transmission.
- The role of Encryption, especially using TLS/SSL.
- Key Management: Challenges in storing and managing secret keys on devices and servers.
- Prevention of Common Attacks: Discussion of attacks such as Replay Attacks, Man-in-the-theMiddle Attacks, and how security measures can prevent them.

9.2.3 Reliable Update Mechanisms

A/B Updates (Dual Bank Updates) Concept:

A detailed explanation of this mechanism which involves having two identical system partitions (Root A and Root B).

Advantages: Ability for safe rollback, and reduced device downtime during updates.

Disadvantages: Doubled storage requirements on the device.

Rollback Mechanisms:

The necessity of rollback in case of update failure or discovery of critical bugs in the new version.

How rollback is performed automatically or manually.

Integrity Checking: Ensuring that the update file has not been corrupted or altered during download.

9.2.4 Flashing Techniques for Microcontrollers Without FOTA

We use the traditional methods to connect directly to the device through a defined interface and there are different types of connections.

9.2.5. Off-Circuit Programming In this technique:

to program the microcontroller, the microcontroller shall be removed from its application circuit, and then connected to a burner, which is a hardware kit that sometimes has a socket on which the microcontroller can be placed, or simply connected to the required pins with jumpers. It uploads the program to the flash memory, and when it finishes, the microcontroller can now go back to its application circuit and start working, hence called off circuit programming as it requires removing the microcontroller from its application circuit.

The flash interface is external outside the microcontroller and the flash memory programming pins are connected to some microcontroller pins to be programmed through. When the file is uploaded to the flash using the burner, the microcontroller can be connected again to its application circuit.

9.2.6. In-Circuit Programming In-system programming (ISP)

also called in-circuit serial programming (ICSP), is the ability of some programmable logic

devices, microcontrollers, and other embedded devices to be programmed while installed in a complete system, rather than requiring the chip to be programmed before installing it into the system.

It also allows firmware updates to be delivered to the on-chip memory of microcontrollers and related processors without requiring specialist programming circuitry on the circuit board and simplifies design work.

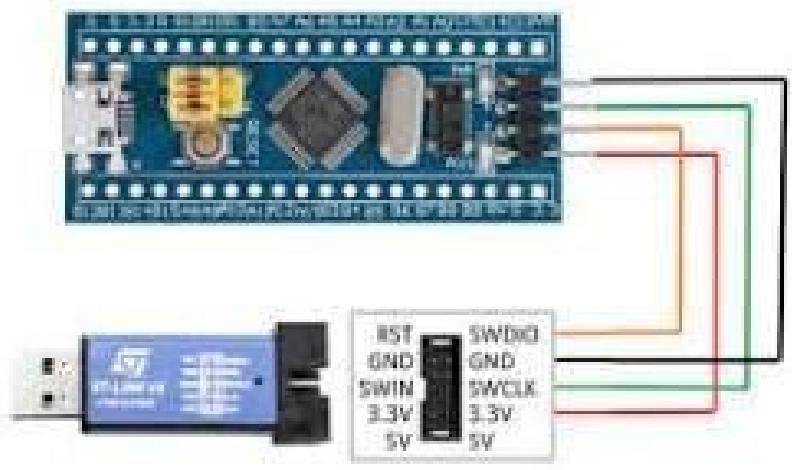
There is no standard for in-system programming protocols for programming microcontroller devices. Almost all manufacturers of microcontrollers support this feature, but all have implemented their protocols, which often differ even for different devices from the same manufacturer.

The primary advantage of in-system programming is that it allows manufacturers of electronic devices to integrate programming and testing into a single production phase, and save money, rather than requiring a separate programming stage before assembling the system. This may allow manufacturers to program the chips in their own system's production line instead of buying pre-programmed chips from a manufacturer or distributor, making it feasible to apply code or

design changes in the middle of a production run. The other advantage is that production can always use the latest firmware, and new features, as well as bug fixes, can be implemented and put into production without the delay occurring when using pre-programmed microcontrollers.

A flash driver is needed to communicate with the external world to get the required application code, this is done by serial communication, so the flash driver manufacturer defines one or more communication protocols through which the flash driver can interface, for example, stm32f103 has 2 protocols that are provided to communicate with the in-circuit flash programmer: JTAG and SWD.

We used the ST-Link programmer and debugger, It works as a translator from the USB protocol to the SWD protocol, this device is target-specific which means that changing the microcontroller to another one with a different flash interface communication protocol will result in changing the translator device, this will be a big headache if it is required to reprogram
several different microcontrollers in the same system, That's one of the reasons why we use a bootloader.



.Figure 9.1

9.3 Mender Tool - Details and Technologies

9.3.1 Introduction to Mender Tool

Mender's philosophy as an open-source solution: providing secure, reliable, and scalable FOTA updates.

Core features: managing Device Groups, Phased Rollouts, and automatic rollback mechanisms.A quick comparison between Mender OSS (Open Source) and Mender Enterprise (commercial version).

9.3.2 Mender Architecture

Mender Server:

Detailed explanation of its components: API Gateway, Device Authentication services, Device Inventory, Deployments Service, and Artifacts Storage.

Underlying technologies: Docker, databases (MongoDB/PostgreSQL), and web server (NGINX).Steps for local setup using Docker Compose.

Mender Client:

Its function and role as an agent on the target device: communicating with the server, downloading and applying updates, and managing failure states.

How to install and configure it on a Linux device.

Mender Artifact:

Mender's specific update package format (.mender): its components, metadata, and actual files.

The importance of Mender Artifacts in ensuring reliable and secure updates

.

9.3.3 Mender Update Process Flow

A step-by-step sequence of events: from artifact building, to uploading to the server, creating a release, deploying the update, and executing it on the device.

The precise interaction between Mender Server and Mender Client (polling for updates, reporting status).

Explanation of different update states (Pending, Downloading, Installing, Rebooting, Success, Failure, Rollback).

9.3.4 Mender Security

Authentication and Authorization: How the connection between the client and server is secured using TLS certificates.Artifact Integrity Verification and Digital Signing: How updates are protected from tampering using digital signatures and hashes.

9. 3.5 Mender Integration with Build Systems:

Yocto Project (Recommended solution for embedded Linux devices):

Explanation of Mender layers in Yocto: meta-mender and meta-mender-core.

Steps to configure Yocto to build Mender-compatible OS images (including A/B partitioning support).

Examples of key configuration files (local.conf, bblayers.conf).

mender-convert (Alternative for Debian/Ubuntu-based systems):

Explanation of the mender-convert tool and how to use it to convert existing OS images into a Mender-updatable format.

Practical usage steps and examples.

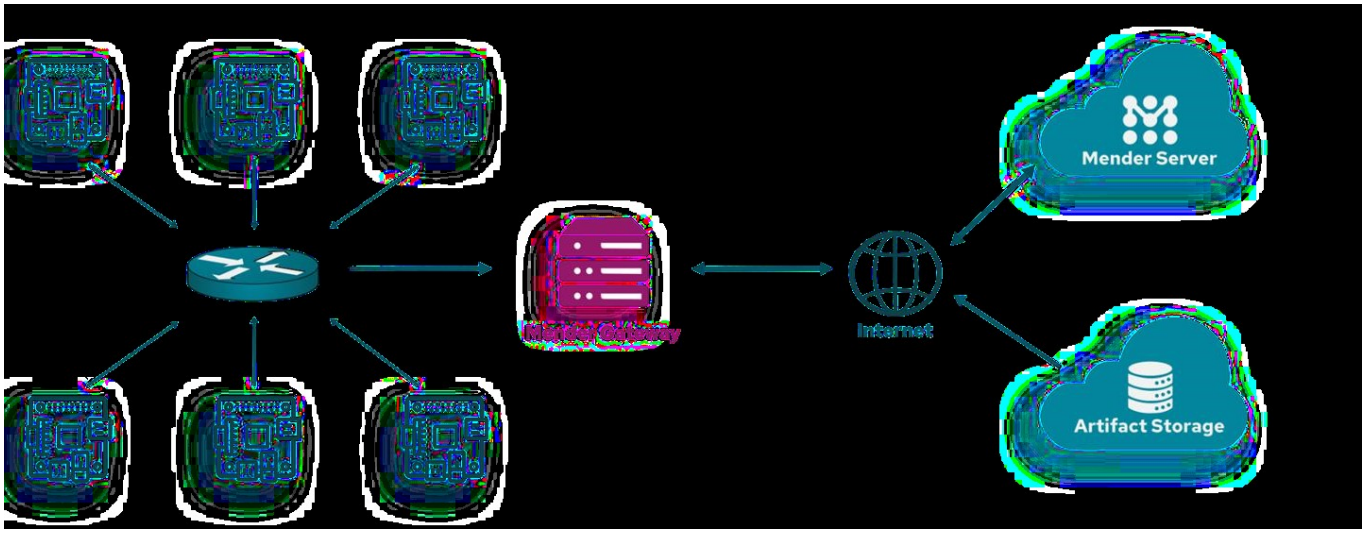


Figure 9.2

9.4 FOTA Implementation on Raspberry Pi using Mender

9.4.1 Setting up the Development Environment

Hardware requirements (Raspberry Pi board, SD Card, development PC).

Setting up the Yocto Project build environment or configuring mender-convert.\

9.4.2 Building a Mender-Compatible OS Image for Raspberry Pi

Using Yocto Project (Preferred method for serious projects): Detailed steps for Yocto setup. Integrating Mender layers and adjusting configuration files to create Raspberry Pi OS images with A/B support and the Mender client.Basic build commands.

Using mender-convert (For quick prototyping):

Steps to apply mender-convert to a Raspberry Pi OS (Debian-based) image for conversion.

9.4.3 Mender Server Setup

Steps for installing Mender Server using Docker Compose on a local machine or cloud server.

Configuring DNS or the /etc/hosts file to access the server.

Navigating the Mender Server User Interface (UI).

9.4.4 Registering the Raspberry Pi with Mender Server

Booting the Raspberry Pi with the Mender-compatible image.

Explaining the initial authentication process and how the device appears in the Mender dashboard.

Steps to accept the device from the server.

9.4.5 Performing a Test FOTA Update on Raspberry Pi

Scenario for creating an update: e.g., changing an application version, adding a new file, or a minor system modification.

Rebuilding the image or converting it to create a new Mender Artifact.

Steps to upload the Artifact to Mender Server and create a new "release."

Deploying the update to the Raspberry Pi through the Mender Server interface.Monitoring the update status directly on the dashboard.

Testing a failure scenario: intentionally causing an update to fail and observing the automatic rollback mechanism.

9.4.6 Results and Performance Analysis

Evaluation of the total update time.

Analysis of network bandwidth consumption.

Review of storage space requirements on the device.

Assessment of the effectiveness of security and rollback mechanisms in tested scenarios.

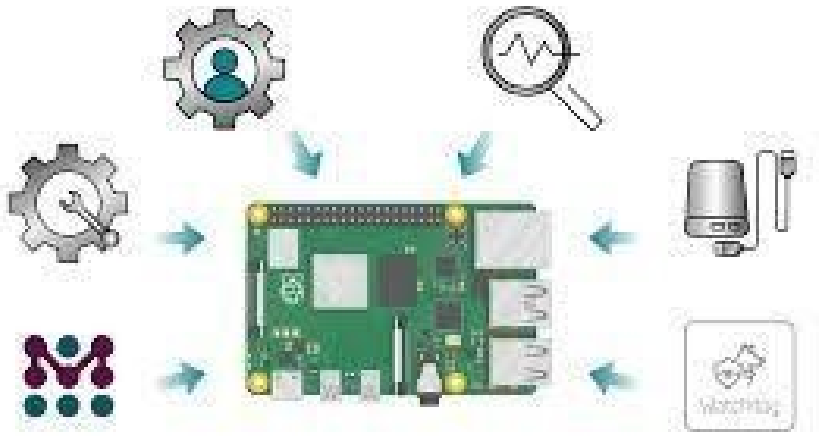


Figure 9.3

9.5 Mender Integration with STM32

Important Note: Direct Mender integration with STM32 as a full "Mender client" is not the typical or recommended scenario, as the Mender Client is designed for embedded Linux systems

A more realistic scenario for a graduation project is to use the Raspberry Pi as a "gateway" or intermediary to update the STM32.

9.5.1 Proposed Use Case Scenario

Raspberry Pi as a Gateway for STM32 Updates:

Architecture explanation: The Raspberry Pi (acting as a Mender client) is the main device.The STM32 (a peripheral device) receives its updates via the Raspberry Pi, not directly from the Mender Server.

Advantages of this approach: Leveraging Mender Server's power and fleet management for the Raspberry Pi, while enabling updates for resource-constrained STM32 devices.

9.5.2 STM32 Update Mechanism via Raspberry Pi

Embedding STM32 Firmware in the Raspberry Pi's Mender Artifact:

Explaining how STM32 .bin or .hex files can be part of the Raspberry Pi's Mender update package.

How the Mender Client on the Raspberry Pi downloads these files as part of its own update .

Developing an Intermediary Application on Raspberry Pi:

Writing a script or application (using Python or C++) on the Raspberry Pi.

Function of this application: Detecting the presence of new STM32 firmware after the Raspberry Pi's own update succeeds.

Communicating with the STM32 via interfaces such as UART, SPI, I2C, or USB.

Initiating the update process on the STM32 and sending the new data.

9.5.3 STM32 Firmware Requirements

Custom Bootloader on STM32:

- Designing and developing a custom Bootloader on the STM32 capable of receiving new firmware from the Raspberry Pi.
- Bootloader functions: erasing flash memory, writing new data, and verifying firmware integrity (using CRC or digital signatures if resources permit).
- Explaining how to switch between Bootloader mode and user application mode. STM32 Update Package Format: This might be binary (.bin) or hex (.hex) files.

9.5.4 Practical Integration Example

Setting up an STM32 development environment (e.g., STM32CubeIDE) and developing the Bootloader and a test application.
Writing a Python script on the Raspberry Pi to manage communication and updates for the STM32.
Integrating this script and the STM32 firmware files into the Raspberry Pi's Yocto/Debian image.
Steps to test the end-to-end update process: a Raspberry Pi update leading to an STM32 update.

9.5.5 Challenges of STM32 Integration and Proposed Solutions

Handling communication protocols between RPi and STM32.
Memory constraints on STM32 for storing the Bootloader and temporary update data.Verifying update integrity on the STM32.

Figure 9.4

9.6 Challenges, Solutions, Results, and Conclusion

9.6.1 Challenges Faced by the Project and Proposed Solutions

Specific technical challenges in setting up the Yocto environment, configuring the Mender Server, or dealing with A/B disk partitioning.
Difficulties in communication between the Raspberry Pi and STM32 or in developing the STM32's custom Bootloader.
Issues related to update security or reliability. How these challenges were overcome.

9.6.2 Analysis of Results

Evaluation of the developed FOTA system's performance based on defined criteria (e.g., update time, resource consumption, effectiveness of security and rollback mechanisms).

Discussion of the extent to which project objectives were achieved.
Comparison of achieved results with initial expectations or goals.

9.6.3 Conclusion

Summary of the project's main achievements and key contributions.
Confirmation of Mender's importance as an effective FOTA solution.
Verification of all project objectives being met.

9.6.4 Future Work

Proposing possible improvements to the system (e.g., enhancing update efficiency, adding new features like delta updates for STM32).Exploring the possibility of integrating other types of devices. Expanding the scope of testing to include more complex scenarios.

9.7 Flow chart

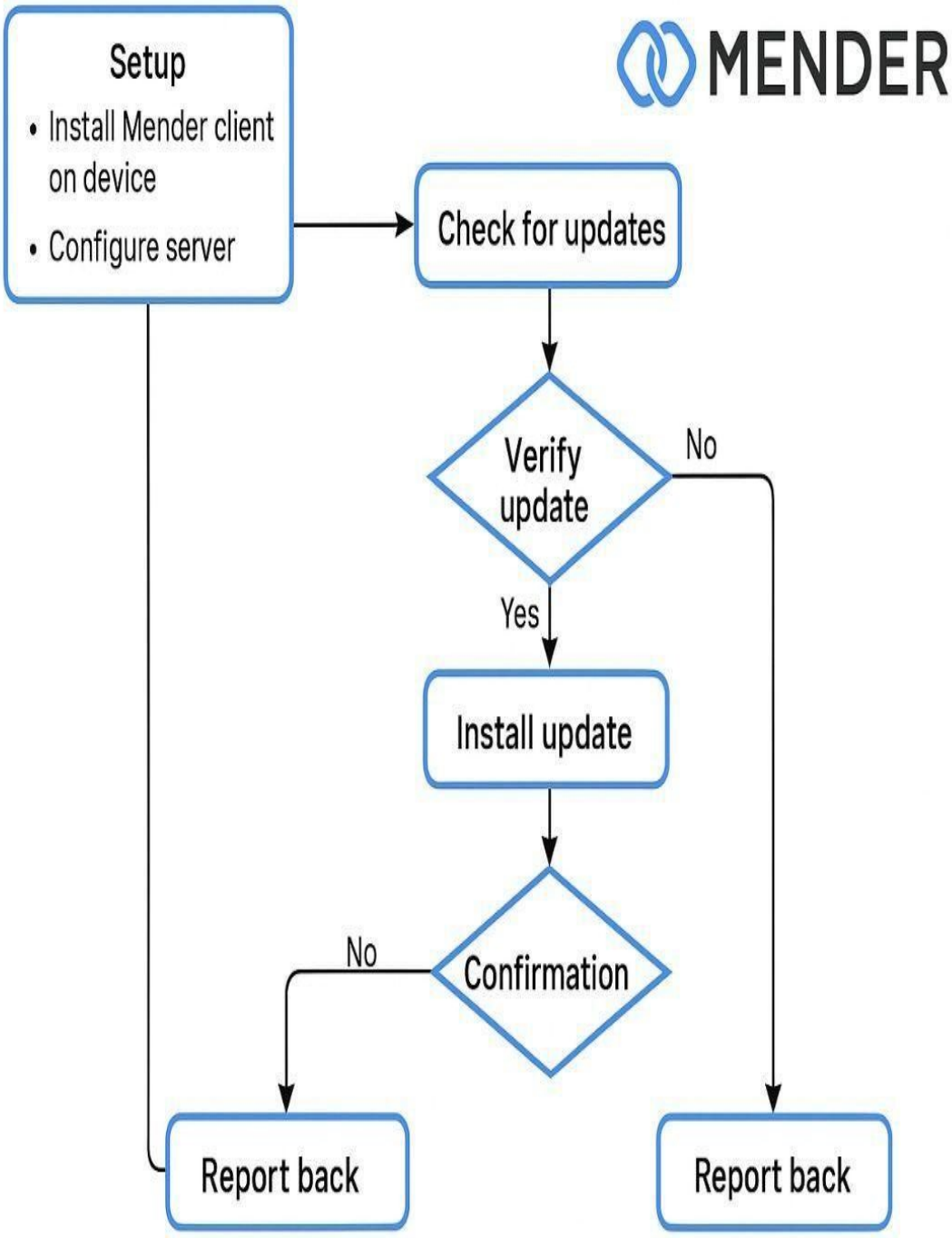


Figure 9.5

CHAPTER 10:
Logger (DTCHandler)

10.1 Introduction to Diagnostic Trouble Codes (DTCs)

In the intricate world of modern vehicles and embedded systems, Diagnostic Trouble Codes (DTCs) are the unsung heroes of effective troubleshooting and maintenance. Think of them as the system's way of telling you, "Hey, something's not quite right here!" These alphanumeric codes are generated by a vehicle's onboard computer (often called an
Electronic Control Unit or ECU) when it detects a fault or anomaly within its vast network of sensors, actuators, and electronic components.

DTCs provide a standardized language for identifying malfunctions, ranging from a simple loose gas cap to more severe engine misfires or transmission issues. Without them, diagnosing problems would be a frustrating, time-consuming, and often impossible task, relying solely on guesswork and extensive manual inspections. Instead, when a fault occurs, the ECU logs a specific DTC, often illuminating a warning light on the dashboard (like the dreaded "Check Engine" light). This allows technicians and even car owners to use diagnostic tools to read the code, pinpointing the general area of the problem and significantly streamlining the repair process.

From ensuring optimal performance and fuel efficiency to guaranteeing safety and reducing emissions, DTCs play a critical role in keeping complex systems running smoothly. They are the backbone of modern diagnostics, transforming a cryptic breakdown into a solvable puzzle. Understanding DTCs, how they're generated, and how they're managed is fundamental to anyone working with contemporary automotive or embedded electronic systems.

10.1.1 What are Diagnostic Trouble Codes (DTCs)?

Definition of DTCs: What they are, why we use them, and their importance in modern automotive and embedded systems.

Examples of common DTCs: (e.g., P0XXX, C0XXX, B0XXX, U0XXX) and their general meanings.

Standard classifications of DTCs: (SAE J2012, ISO 15031-6).

How DTCs are generated: The error detection process by the Electronic Control Unit (ECU).

Types of DTCs:

1.Active
2. Pending
3. History/Stored
4. Permanent.

10.1.2 The Need for DTC Management Systems

Challenges in handling DTCs manually.Importance of quick and accurate problem diagnosis.

The role of DTC management software in simplifying maintenance and repair.

Operational and economic benefits of an effective DTC management system.

10.1.3 Overview of DTCHandler's Role in the Project

DTCHandler's position within the overall project architecture.

The problem DTCHandler solves: (in-memory DTC management).

Key design goals for DTCHandler: (flexibility, efficiency, scalability).

Figure 10.1

10.2 DTC Handler Design and Architecture

10.2.1 Functional and Nonfunctional Requirements

Functional: Registering DTCs, updating them, retrieving them, deleting them, emitting signals upon registration.

Non-functional: Performance (response speed), reliability (no data loss during operation), scalability (handling a large number of DTCs), usability (clear API), maintainability.

10.2.2 Code (Header File - DTCHandler.hpp)

```
#ifndef DTCManager_HPP
#define DTCManager_HPP

#include <QObject>
#include <QString>
#include <QMap>

class DTCManager : public QObject
{
    Q_OBJECT
    Q_CLASSINFO("D-Bus Interface", "com.car.DTCManager.Interface")
public:
    DTCManager(QObject *parent = nullptr);
    bool ExportToDBus();
    ~DTCManager();

public slots:
    void logDTC(const QString &code, const QString &description);
    QMap<QString, QString> getActiveDTCs() const;
    void clearDTC(const QString &code);

signals:
    void newDTCLogged(const QString &code);
private:
    QMap<QString, QString> m_activeDTCs;
};

#endif // DTCManager_HPP
```

Figure 10.2

10.2.3 Technology Selection:

Why Qt?

1.Introduction to the Qt framework.

2.Advantages of using Qt:(objects, signals/slots, data structures, ease of development).

- **Why QObject?**
  -The role of the Meta-Object System.
- **Why QString?**
  -Its advantages over C++ std::string.
- **Why QMap?**
  -Explanation of QMap as a data structure (compared to QHash or std::map), its efficiency in searching, adding, and deleting.

10.2.4 Interface Design (Header File - DTCHandler.hpp)

Analysis of each class member:Constructor and Destructor: Their roles, and why there's no database closing logic.

logDTC(): Explanation of parameters and function, how it handles new registration and updates.

getActiveDTCs(): Its function, reason for returning a const copy. clearDTC(): Its function. newDTCLogged signal: Importance of signals in Qt, and how they allow for decoupling.

10.2.5 Code (Source File - DTCHandler.cpp)

```cpp
#include <QDebug>
#include "DTCManager.hpp"
#include "../dbusManager/dbusManager.h"

DTCManager::DTCManager(QObject *parent) : QObject(parent) {
    qDebug() << "DTCManager initialized. Using in-memory QMap for DTCs.";
}


DTCManager::~DTCManager() {
    qDebug() << "DTCManager destroyed.";
}

void DTCManager::logDTC(const QString &code, const QString &description) {
    if (m_activeDTCs.contains(code)) {
        qDebug() << "DTC Code" << code << "already exists. Updating description.";
        m_activeDTCs.insert(code, description);
    } else {
        m_activeDTCs.insert(code, description);
        emit newDTCLogged(code);
        qDebug() << "New DTC logged: Code=" << code << ", Description=" << description;
    }
}

QMap<QString, QString> DTCManager::getActiveDTCs() const {
    return m_activeDTCs;
}

void DTCManager::clearDTC(const QString &code) {
    if (m_activeDTCs.remove(code)) {
        qDebug() << "DTC removed: Code=" << code;
    } else {
        qDebug() << "DTC Code" << code << "not found.";
    }
}

bool DTCManager::ExportToDBus() {
    DBusServiceInfo bus_info = {
        .service_name = "com.car.DTCManager",
        .object_path = "/com/car/DTCManager",
    };

```

Figure 10.3

10.2.6 Implementation Design (Source File - DTCHandler.cpp)

Detailed implementation of each function:

- DTCHandler::DTCHandler() and~DTCHandler():
  Initialization and cleanup.

- logDTC(): Step-by-step explanation of the logic, focusing on the contains() condition to emit the signal only on new additions, and how the update process works.
- getActiveDTCs(): Explanation of how return by value works and its impact.
- clearDTC(): Explanation of remove() usage and how to check for success. Handling edge cases: such as an empty code

.

10.3 Using and Testing DTC Handler

10.3.1 Basic Usage Scenarios

Full code example (main.cpp) demonstrating:

- Object creation.
- Registering new DTCs.
- Updating existing DTCs.
- Getting and displaying the list of DTCs.
- Clearing DTCs.
- Handling clearing a non-existent DTC.
- Detailed explanation of each step in main.cpp.

10.3.2 Unit Testing for DTCHandler

Importance of unit testing in software development.

How to write tests using Qt Test (or another testing framework if preferred).

Examples of test cases for:

- Registering a new DTC.
- Updating an existing DTC.
- Getting an empty DTC list.
- Getting a DTC list after addition.
- Clearing an existing DTC.
- Clearing a non-existent DTC.
- Testing the newDTCLogged signal emission (can use QSignalSpy).

10.3.3 Integrating DTCHandler into Larger Applications

How can a Graphical User Interface (GUI) interact with DTCHandler?
Connecting the newDTCLogged signal to a slot in the UI to automatically update the display.DTCHandler's role in the Business Logic Layer of a larger application.

10.4 Future Expansion and Enhancements

10.4.1 Data Persistence

The problem: Data loss when the application closes.
Proposed solutions:

1. Storing in a file: (JSON, XML, CSV).
   Explanation of how to read and write data in these formats using Qt(e.g., QJsonDocument, QXmlStreamWriter).
2. Design of saveDTCsToFile() and loadDTCsFromFile() functions. Storing in an SQLite database:

Why SQLite? (Lightweight, no server required).
How to use Qt SQL Module (QSqlDatabase, QSqlQuery)? Database schema design (DTCs table).Modifying DTCHandler to interact with the database directly instead of QMap, or adding a separate persistence layer.

10.4.2 Multithreading Considerations

When multithreading becomes an issue: (concurrent access to m_activeDTCs).
Qt solutions for synchronization: QMutex. How to use QMutexLocker to protect operations on m_activeDTCs.

10.4.3 Additional Features and Proposed Improvements

1. DTC Filtering: By type, priority, or date.
2. DTC Archiving: Moving old or "resolved" DTCs to an archive log.
3. DTC State Management: Adding a property to track DTC status (e.g., "Active," "Processed," "Cleared").

Configuration: Loading DTC rules from an external file.
Integration with External Systems: How to send DTCs to a central logging system or a cloud platform.

Figure 10.4

10.5 System Flow chart

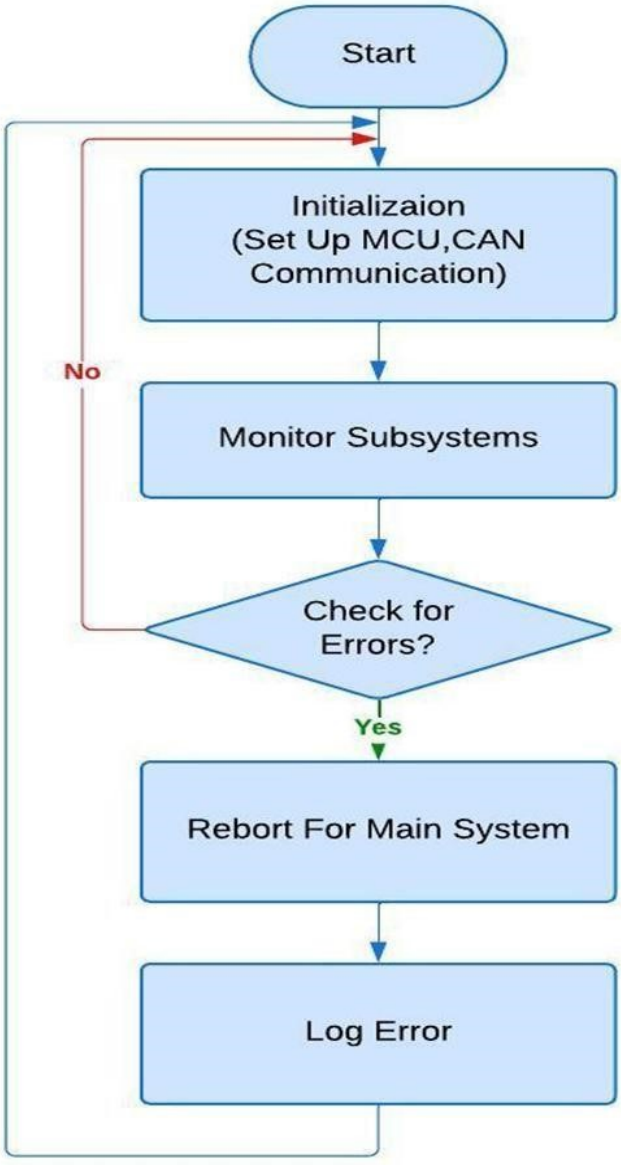


Figure 10.5

Module D

Embedded Linux

Chapter 11 : Embedded Linux.
Chapter 12 : Embedded Linux Development.
Chapter 13: Main System (GUI)
Chapter 14: Embedded Linux Development Using Yocto project.
Chapter 15: Programming Enhancements
Chapter 16: Future work and conclusion

CHAPTER 11:
Embedded Linux

11.1 Introduction

Linux first became a viable choice for embedded devices around 1999. That was when Axis (https://www.axis.com) released their first Linux-powered network camera and TiVo (https://business.tivo.com) their first Digital Video Recorder (DVR).

Since 1999, Linux has become ever more popular, to the point that today it is the operating system of choice for many classes of product. In 2021, there were over two billion devices running Linux.

That includes a large number of smartphones running Android, which uses a Linux kernel, and hundreds of millions of set-top boxes, smart TVs, and Wi-Fi routers, not to mention a very diverse range of devices such as vehicle diagnostics, weighing scales, industrial devices, and medical monitoring units that ship in smaller volumes.

11.2 Linux

11.2.1 Linux History

Linux is a family of open-source Unix-like operating systems based on the Linux kernel, which was first released by Linus Torvalds on September 17, 1991. The development of Linux has a rich history intertwined with the evolution of computer technology and the open source movement.

- **Key Milestones in the History of Linux:**
  - **1983: GNU Project**

    Richard Stallman started the GNU Project to develop a free Unix-like operating system. The project created many essential components, but lacked a kernel.
  - **1987: MINIX**

Andrew S. Tanenbaum released MINIX, a Unix-like operating system designed for educational purposes. MINIX inspired many, including Linus Torvalds.

- **1991: Linux Kernel Announcement**

  Linus Torvalds, a Finnish student, announced the development of a new kernel on the comp.os.minix newsgroup. This kernel, combined with GNU tools, formed a complete operating system.

  Torvalds released version 0.01 of the Linux kernel on September 17, 1991, and version 0.02 on October 5, 1991, under a custom license.

- **1992: Linux Becomes Open Source**

  Torvalds relicensed Linux under the GNU General Public License (GPL), which allowed the broader community to freely modify and distribute the software.

- **1993: Early Distributions**

  The first Linux distributions, including Slackware and Debian, were released, simplifying installation and management of Linux systems.

- **1994: Linux Kernel 1.0**

  The Linux kernel reached version 1.0, marking a significant milestone in its development.

- **Late 1990s: Commercial Interest and Development**

  Major companies like IBM, Compaq, and Oracle began to support Linux, recognizing its potential in enterprise environments.

  The open-source community developed key components like the Apache web server, contributing to Linux's popularity.

- **2000s: Desktop and Server Growth**

  Linux gained popularity as a server operating system and started making inroads into the desktop market with distributions like Ubuntu, which aimed to be user-friendly.

- **2010s: Expansion and Dominance**

  Linux became the backbone of many technologies, including Android (based on the Linux kernel), powering the majority of smartphones.

  Linux's presence in data centers, cloud computing, and supercomputing grew significantly.

- **2020s: Continued Evolution**

Linux continues to evolve with contributions from a vast community of developers worldwide.

It plays a crucial role in emerging technologies like IoT, AI, and edge computing.

- **Influential Linux Distributions:**

Nowadays, there are many distributions of GNU/Linux systems such as:

- **Debian:** Known for its stability and robustness, Debian is the foundation for many other distributions, including Ubuntu.
- **Red Hat Enterprise Linux (RHEL):** A commercially supported distribution that is popular in enterprise environments.
- **Ubuntu:** Based on Debian, it focuses on ease of use and has both desktop and server versions.
- **Fedora:** Sponsored by Red Hat, Fedora is a cutting-edge distribution that serves as a testing ground for new features.
- **Arch Linux:** Known for its simplicity and customization, it is aimed at advanced users.

Combining Linux with the almost-complete GNU system resulted in a complete operating system: GNU/Linux system.

11.2.2 Linux History

Why is Linux so pervasive? And why does something as simple as a TV need to run something as complex as Linux just to display streaming video on a screen?The simple answer is Moore's Law: Gordon Moore, co-founder of Intel, observed in 1965 that the density of components on a chip will double approximately every 2 years.

That applies to the devices that we design and use in our everyday lives just as much as it does to desktops, laptops, and servers. At the heart of most embedded devices is a highly integrated chip that contains one or more processor cores and interfaces with main memory, mass storage, and peripherals of many types.

This is referred to as a System on Chip, or SoC, and SoCs are increasing in complexity according to Moore's Law. A typical SoC has a technical reference manual that stretches to thousands of pages. Your TV is not simply displaying a video stream as the old analog sets used to do.

The stream is digital, possibly encrypted, and it needs processing to create an image. Your TV is (or soon will be) connected to the internet. It can receive content from smartphones, tablets, and home media servers. It can be (or soon will be) used to play games and so on. You need a full operating system to manage this degree of complexity.

Here are some points that drive the adoption of Linux:

- Linux has the necessary functionality. It has a good scheduler, a good network stack, support for USB, Wi-Fi, Bluetooth, many kinds of storage media, good support for multimedia devices, and so on. It ticks all the boxes.
- Linux has been ported to a wide range of processor architectures, including some that are very commonly found in SoC designs – Arm, MIPS, x86, and PowerPC.
- Linux is open source, so you have the freedom to get the source code and modify it to meet your needs. You, or someone working on your behalf, can create a board support package for your particular SoC board or device. You can add protocols, features, and technologies that may be missing from the mainline source code. You can remove features that you don't need to reduce memory and storage requirements. Linux is flexible.
- Linux has an active community; in the case of the Linux kernel, very active. There is a new release of the kernel every 8 to 10 weeks, and each release contains code from more than 1,000 developers. An active community means that Linux is up to date and supports current hardware, protocols, and standards.
- Open source licenses guarantee that you have access to the source code. There is no vendor tie-in.

For these reasons, Linux is an ideal choice for complex devices. But there are a few caveats I should mention here. Complexity makes it harder to understand. Coupled with the fast moving development process and the decentralized structures of open source, you have to put some effort into learning how to use it and to keep on re-learning as it changes.

11.2.3 Difference between Linux and Windows

Feature	Linux	Windows
Source Code	Open Source	Closed Source
Licensing	GNU General Public License (GPL)	Proprietary License
User interface	Multiple desktop environments (GNOME, KDE, XFCE, etc.)	Consistent, standardize interface
Software availability	Strong support for open source software, fewer games	Extensive commercial softwa and game support
Performance & Stability	Efficient on older hardware, known for stability	Requires more resources, may have stability issues with updates
Security	More secure, less targeted by malware	Frequently targeted by malware, regular security updates needed
Use case	Servers, supercomputers, developers, tech enthusiasts	General desktop use, businesses, education, gaming
Package management	Various package managers (APT, YUM, etc.)	centralized software distributio (Microsoft Store)

Customization	Highly customizable (kernel, desktop environment)	Limited customization, mainly through third-party tools
Cost	Free to use and modify	Requires purchasing a license
Community Support	Strong community support, active forums	Official support from Microsoft, professional support available
Hardware Compatibility	Broad support, but may require manual configuration	Wide range of hardware support, plug and-play
File System	Ext4, Btrfs, XFS, and others	NTFS, FAT32, exFAT
Command Line Interface (CLI)	Powerful and integral (Bash, Zsh)	Command Prompt and PowerShell, less integral for average users

11.3 What is Embedded Linux?

Embedded Linux is a type of Linux operating system/kernel that is designed to be installed and

used within embedded devices and appliances. In other words, it's a compact version of Linux that offers features and services in line with the operating and application requirements of the embedded system. Although it uses the same kernel, Embedded Linux is quite different from the standard operating system. First of all, it gets tailored for embedded systems and, therefore, is much smaller in size, requires less processing power, and has minimal features compared to that of a fully fledged operating system. The Linux Kernel is modified and optimized as an embedded Linux version. Such a Linux instance can only run applications created specifically for the device.

Embedded Linux also offers its developers with several advantages over other systems such as:

1. Cross-Compilation for any supported platform.
2. Community reflection of common vulnerabilities and exposures (CVE) fixes in updated releases.
3. Deployment to commonly used Linux infrastructure and tools.
4. Modern, cloud-native environment.
5. Broad hardware support.
6. Productive lifecycle through community LTS.

11.4 Embedded Linux vs. Bare metal vs. RTOS

Embedded Linux	**Bare-metal**	**RTOS**
Large overhead compared to other technologies due to scheduler, memory management, background tasks, etc.	Little to no software overhead	Scheduler overhead
Requires a microprocessor with a memory management unit (MMU) and an external RAM.	Low power requirement.	Requires a more powerful microcontroller.
Low direct control of hardware (files or abstraction layers).	High control of hardware.	High control of hardware
Multiple threads and processes.	Single-purpose or simple applications, hardware- dependent	Multi-threading.

Multiple complex tasks, like networking, filesystem, graphical interface, etc.	Strict timing.	Multiple tasks, like networking, user interface, etc.

11.5 Embedded Linux Architecture

At the most basic level, an Embedded Linux system is one that uses Linux as the operating system that sits between hardware and the application of an embedded device. There are five key components to an Embedded Linux system which we will go through, and these components are:

- ToolChain
- Bootloader
- Kernel
- Root filesystem
- Services

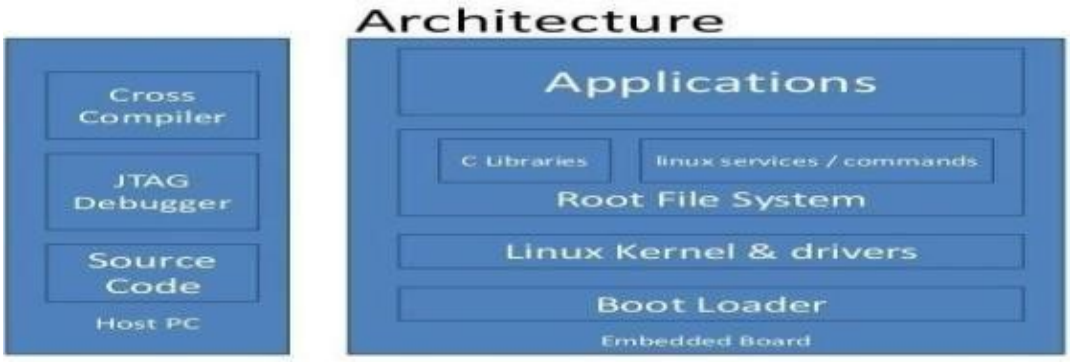


Figure 11.1

11.5.1 Toolchain

A toolchain is a collection of development tools, such as GCC compiler, C libraries, and GNU debugger, which are used to create source code for the target embedded hardware.

There are two types of toolchain:

- **Native toolchain:** This toolchain runs on the same type of system, sometimes the same actual system, as the programs it generates. This is the usual case for desktops and servers.
- **Cross toolchain:** This toolchain runs on a different type of system than the target, allowing the development to be done on a fast desktop PC and then loaded onto the embedded target for testing Normally, cross-compilation toolchains are utilized in embedded Linux, because the target embedded processor does not have sufficient resources for these toolchains. In cross-compilation systems, development is performed on the host system (a powerful PC), and then source code is deployed on the target device using different interfaces such as serial, ethernet, etc. Serial communication is the most commonly used interface between the host system and a target device.

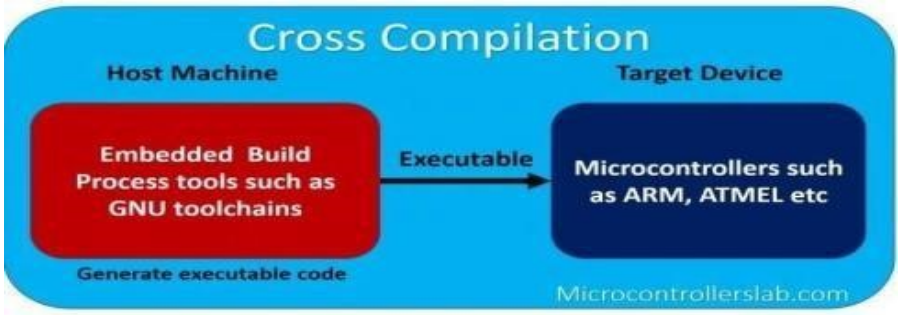


Figure 11.2

Toolchain consists of:

- **Binutils:** is a set of tools to generate and manipulate binaries for a given CPU architecture. ex: as, ld, ar, and ranlib.
- **Kernel headers:** The C library and compiled programs need to interact with the kernel. For example, to get available system calls and their numbers or for available data structures.

  That's why compiling the C library requires kernel headers, and many applications also require them.
- **GCC:** GNU Compiler Collection, the famous free software compiler. Can compile C, C++, Ada, Fortran, Java, Objective-C, Objective-C++, and generate code for a large number of CPU architectures, including ARM, AVR, Black n, CRIS, FRV, M32, MIPS, MN10300, PowerPC, SH, v850, i386, x86 64, IA64, and Xtensa.
- **C library:** The C library is a very important component of a Linux system Interface between the applications and the kernel. It provides the well known

standard C API to ease application development. Several C libraries are available: glibc, uClibc, eglibc, dietlibc, newlib, etc. The choice of the C library for our toolchain must be made at the time of the cross-compiling toolchain generation.

There are three choices for our cross-development toolchain: a ready-built toolchain that matches our needs, use the one generated by an embedded build tool, selecting a Build System, or can create one using crosstool-NG.

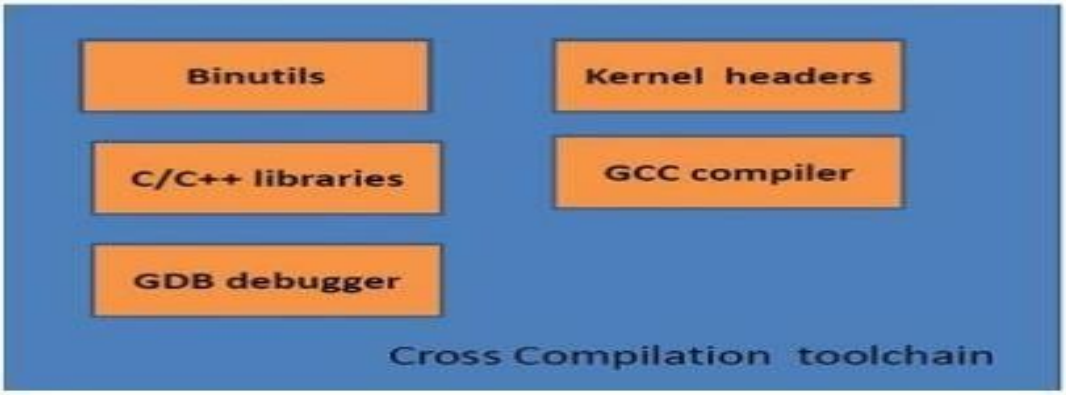


Figure 11.3

11.5.2 Bootloader

This is a piece of code that runs when we apply power to the embedded hardware for the first time.Bootloader performs various hardware initialization steps before loading the operating system. When a processor boots up or resets, each processor has a number of predefined hardware steps and configurations.

The followings are the main roles of a bootloader:

- Target hardware initialization.
- Load the application binary (operating system, root file system, device tree) from the nonvolatile external storage to the RAM memory.
- RAM connects externally to the SOC and it stores run-time data, operating system, root file system, application software, and stack/heap.
- After that, execute the main operating system code from the RAM memory.
- Different microprocessors may have a distinct process of booting sequence but the underlying concepts remain the same.
- In embedded Linux, two generic or universal bootloaders are popular such as Uboot and busybox.

11.5.2.1 Embedded Linux Boot flow Process

What exactly are the steps and what is required to successfully boot the Linux kernel of embedded Linux boards such as BeagleBone, Raspberry pi, etc? If we take the example of Beaglebone Black, this board has an AM335x SoC manufactured by Texas Instruments. In order to boot Linux on this Soc following are the four main components as shown in the figure below:

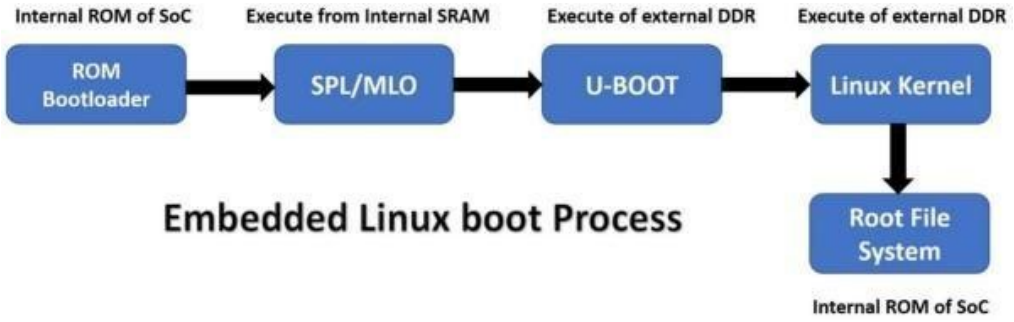


Figure 11.4

- The first component is a first stage bootloader or a ROM bootloader. This is usually stored inside the flash memory of the SoC by the manufacturer and we cannot change it.
- The main role of RBL is to load the second stage bootloader that is SPL (a secondary program loader) or MLO (memory loader) from the internal SRAM of SoC.
- The job of MLO is to load and execute a third stage bootloader such as U-Boot from the DDR memory of the development board but DDR is off the SoC.
- The third stage bootloader which is U-boot loads the Linux kernel from persistent storage such as an SD card to DDR memory of the development board.
- After that system starts to execute its first process initi and mounts the root file system.

11.5.3 Linux Kernel

It is a software or operating system kernel that manages the resources of embedded processors optimally and efficiently. The main core component of embedded Linux is the Linux Kernel. It is a layered operating system architecture, which means it is divided into two layers: user space and kernel space memory space.

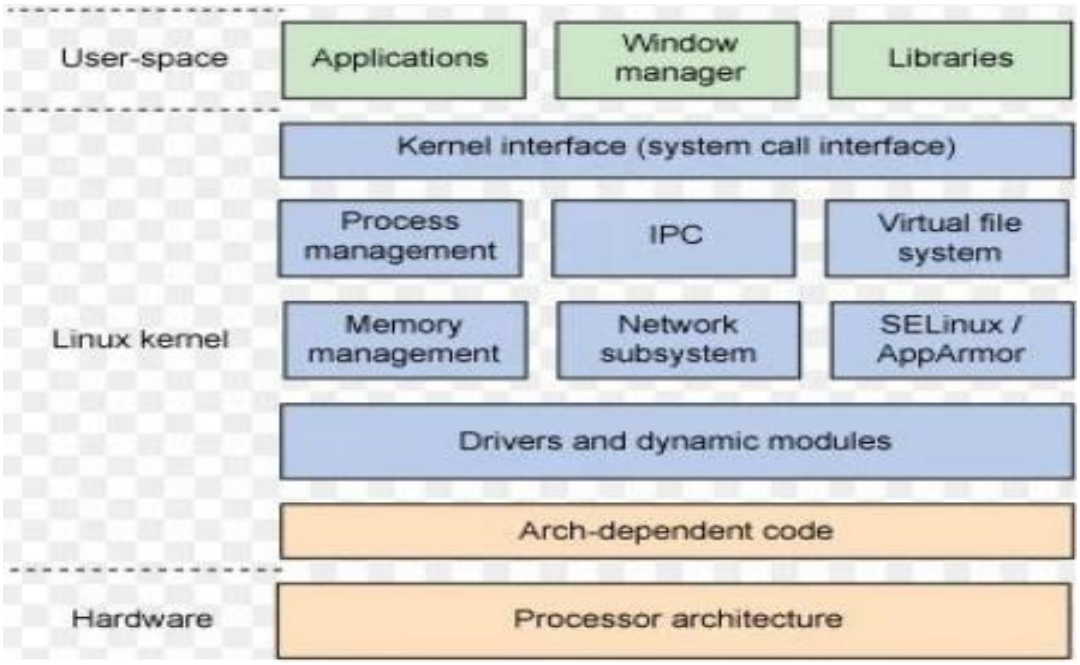


Figure 11.5

It provides various services such as:

- Process management (Allocates CPU resources concurrently to processes/tasks through scheduler)
- Process scheduling (Schedules concurrently running process to run on the CPU using scheduling algorithms)
- Memory management (Manage memory for different processes through virtual memory support)
- Interprocess communication (Sometimes tasks require to share data with each other and the interprocess communication feature provides this support through shared memory)
- File management (Manage how information is stored and accessed by the system)
- I/O management (Supported through device drivers such as character, block, and USB device drivers)
- Networking (Support for network device drivers) ● Interface with hardware through device drivers.

To add and remove code from the Linux kernel at runtime we use a machine called loadable kernel module. They are responsible for communication between kernel

and hardware without having to know the functionality of hardware and they are perfect for the drivers of the device. The kernel module runs in kernel space. Memory addresses of kernel space and user space are unique and they don't overlap. This methodology shows that the applications that run in user space have a reliable view of hardware, irrespective of the platform of hardware.

11.5.4 Device Tree

The embedded Linux-based hardware platforms also contain peripherals that do not have the ability to acknowledge their existence to the operating system kernel. There must be a way to inform the Linux kernel about these hardware peripherals.

The Linux kernel contains thousands of device drivers, and during the boot-up process, the kernel should know which non-discoverable hardware devices are connected to your embedded Linux-based system. The question is, how does the kernel find the details of the connected hardware devices? The answer is the device tree.

A device tree is a tree data structure that provides information about the connected hardware devices that are not discoverable themself and other on-chip peripherals of an SoC. Some of the automatically non-discoverable on-chip peripherals are I2C, UART, SP, ethernet, etc. On the contrary, the USB device has the capability or intelligence to inform its presence to the operating system dynamically.

DT describes these devices using a special hardware description language. It decides which drivers to initialize and configure the device parameters such as registers addresses and IRQs. A DT source is combined with the Device Tree Blob and combines with the kernel during the embedded hardware booting process. The bootloader also loads the Device Tree Blob into RAM memory along with the Kernel and it runs before the kernel. In short, DT manages hardware resources of embedded Linux based systems.

11.5.5 Root File systems

In Linux, Filesystems organize data into files and directories inside the storage device. In fact, everything in Linux is a file, stored, and accessed through a file. These files and directories are organized in a hierarchical form.

During the boot-up process, the Linux kernel needs one more component: a root filesystem. A root filesystem is a special type of filesystem that contains all the configuration files required to prepare the execution environment for the embedded device. For example, it prepares graphical displays, and it also contains the first parent process running on Linux (init).

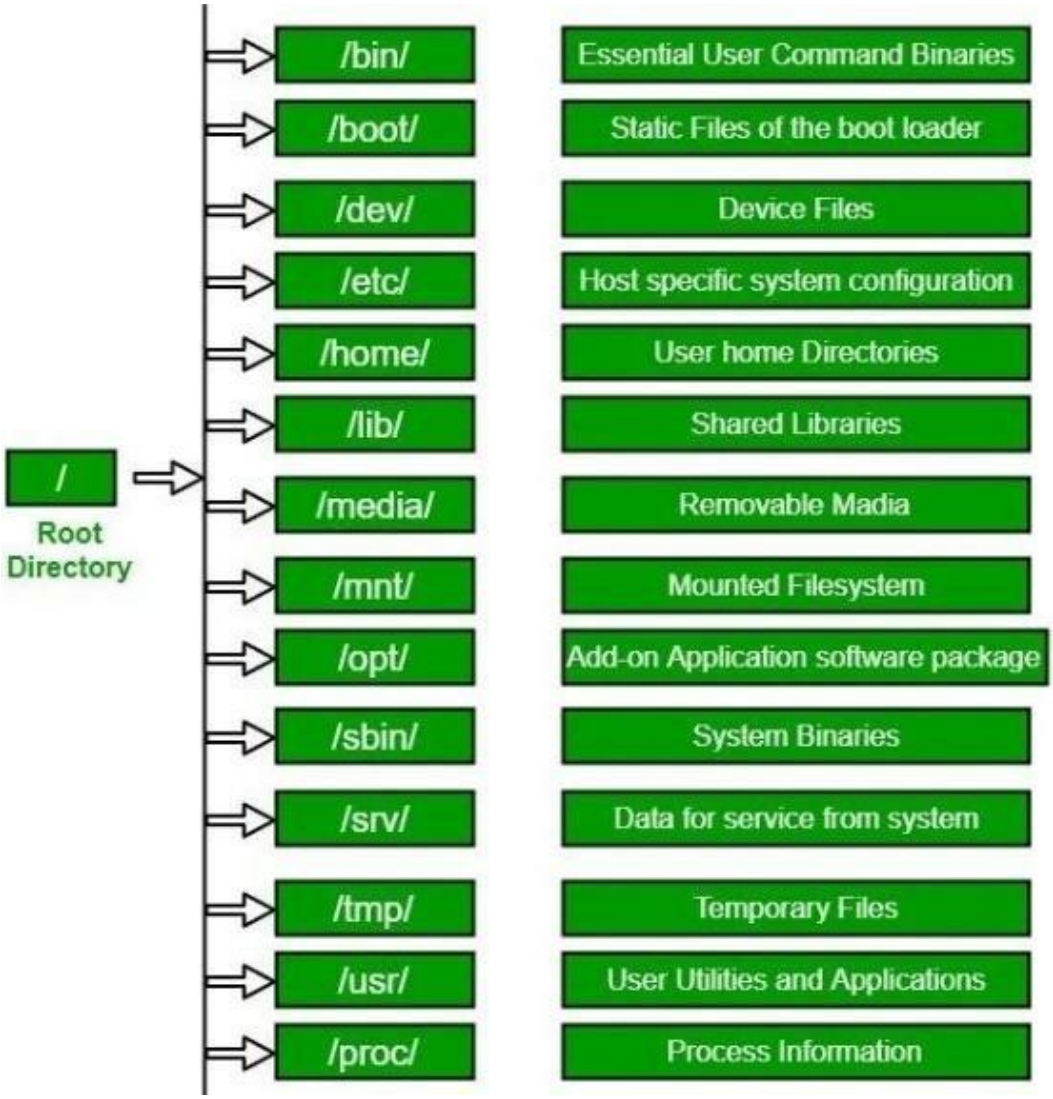


Figure 11.6

The Linux kernel works with the root file system. This is the file system onto which the root can be mounted. There are two types of file systems for desktop and server distributions.

These are the initial root system and the real root system. The initial root system is used for

mounting and running the latter. The initial root system is used to load drivers on a desktop. The switch between the initial root file system and the real root file system is called a pivot. The first objective is to create a minimal root filesystem that can give us a shell prompt. Then using that as a base we will add scripts to start other programs up, and to configure a network interface and user permissions. Knowing how to build the root filesystem from scratch is a useful skill. To make a minimal root filesystem, we these components:

- Init
- Shell
- Daemons
- Shared libraries
- Configuration files
- Device nodes
- Kernel modules

11.5.6 Applications

Embedded Linux enables us to run programs in higher level language than that of bare metal embedded. Languages like python, C/C++, and Rust are the most common. The init program is responsible for starting these programs.

CHAPTER 12:
Embedded Linux Development

12.1 Introduction

Embedded Linux is a wide field as it can be seen from the previous chapters, hence an Embedded Linux developer has many goals, targets and responsibilities, all which can be divided to:

- Develop and maintain C/C++, python, or rust codes.
- Develop and maintain Linux device drivers.
- Develop and maintain Linux environment either through Yocto (most common), build root, or any other method.

In this Chapter we will discuss the development process of an Embedded Linux Image.

12.2 SD card structure in Embedded Linux

Before we dive any deeper, let's first talk about the structure of the SD card we will be using on our target device. Recall the Embedded Linux elements we discussed earlier:

- ToolChain
- Bootloader
- Kernel
- Root filesystem
- Services

Each element should be placed in a specific manner, but first we need to part out own SD card into two partitions:

➢Boot, which is a small FAT partition, will contain our bootloader, the kernel, device tree binary (DTB), and the configurations.

➢Rootfs, which is a large EXT4 partition, will contain our directories (/user, /bin, etc)

Now, we need to understand a very important term that we'll see very often not only when dealing with Embedded Linux, but with Linux in general, and that is 'Mount'.
The data we place on our SD card is stored in binary, so how are these ones and zeros transformed into actual information? The kernel contains a virtual file system (VFS) responsible for mapping binaries into readable data (.txt, .img … etc.). This process is called 'Mounting', where a row of data is transformed into readable data. If we were to try to access or read the data without mounting, the output data would be garbage values.

12.3 The Traditional Method for building a Customized Linux Image

Rather than the traditional bare metal embedded software development, where we write a C code, build, and produce a hex file, then burn the hex file on our target hardware), we create a Linux image that is flashed on the development board through an SD card.
The traditional method for building a Linux image is as it may sound, more of a primitive approach where we must, as they say, roll up our sleeves and do all the work by ourselves. The development process can be broken down to the following steps:

1. Study the board of choice, and specifically the booting process.
2. Build or download a toolchain.
3. Download a bootloader, which is a special piece of software with one sole purpose and that is to load the kernel and hand over the control to it (U-boot is the most popular one).
4. Build the kernel, which is the heart of the operating system.
5. Build the filesystem, depending on our configurations.
6. Build our application
7. Plug and play

12.3.1 Studying the booting sequence

In the previous chapters, we covered the booting sequence of Raspberry Pi with all its steps and elements.

12.3.2 Build or download a Toolchain

First, what is meant by a toolchain, a toolchain is a set of tools that compiles source code into executable that can run on our target device (includes a compiler, kernel headers, binutils, a linker, and run-time libraries).

In order to discuss the importance of the toolchain, we must first understand an important concept, and that is the difference between a cross compiler and a native Compiler.

Let's say that we have a c file called test.c, we compile it using GCC compiler to output test.exe that we run on our host machine. In this case, GCC is called a native compiler because we used it to generate code for the same platform on which it runs.

Say that for the same c file called test.c, we used arm.gcc to compile and output an executable that will run on an ARM target device. In this case, arm.gcc is called a cross compiler because it was used to generate executable code for a platform other than the one on which the compiler is running.

Although we could simply install the compiler on RPI using sudo apt-get but we don't want to do this, we instead want to have/build our own toolchain for better customization. Of course, we are not going to build everything from scratch, but instead we will be using tools like crosstool-ng.

1. Clone the repository.
2. Search for suitable configurations.
3. Set the default configurations according to the board we're using.
4. Apply all the necessary edits through menuconfig (Kconfig, which is a GUI that will provide us with helpful tools to output the .config file).
5. Build.

Note that the above sequence will be used for developing all elements, not just the toolchain.

12.3.3 Building a bootloader (U-boot)

Let's first take a look at U-boot or the Universal Boot Loader and understand what it is. It supports a wide range of microprocessors like MIPS, ARM, PPC, Blackfin, AVR32 and x86. It also supports different methods of booting, which is neat, it can support booting from USB, SD card, NOR and NAND flash (non-volatile memory).

It can also boot the Linux kernel from the network using TFTP. U-boot also provides a command line interface which gives us very easy access to it and many different things.
As we discussed in the previous chapters, U-boot is the second stage bootloader, responsible for loading the kernel and giving it control. When U-boot is compiled, we get u-boot.img.
To use U-boot on our target device, we will follow the same steps as in the toolchain.
First, we will download the source code, which is of course open source and available on GitHub. We will then identify then specify the architecture we will be using, which can be found either through the target device's name or the SOC in the config file. Now that we have specified the architecture, we also need to identify what family it's under (for example, if we were to use this method, our architecture goes under the ARM family).

Running the make command followed by the architecture and family will output the .config file. The next step is to configure the U-boot configurations through menuconfig. After applying all necessary configurations, all that is left now is to build. So, we simply run the build command followed by the number of cores to utilize through the build process.
After building, we should now have the u-boot.bin file.

12.3.4 Building a Linux Kernel

First of all, a kernel can be a zimage or a uimage but it shouldn't really matter for us as all of them are just different ways to store. Recall the function of a kernel in Embedded Linux:

- Runs and schedules different processes.
- Manages resources.
- Manages memory.
- Handles interrupts and multi users.
- Provides filesystem, networking and security

The kernel is, as one could imagine, a bunch of code (written in C) compiled to binary. This binary
output which resembles the kernel doesn't have any information about the hardware configurations. Instead, these configurations can be found in the Device Tree Binary

(DTB). Note that DTB files were at first stored as Device Tree Source files (DTS) which were then compiled using a Device Tree Compile (DTC).

We will follow the same sequence we used in the toolchain and bootloader. First, we will download the source code through Github, as usual. Then we will specify the target device architecture, the family, and enable the cross compiler, all through the build command. All that is left now is to apply all the necessary configurations through menuconfig, as we did before, and to finally build.

12.3.5 Building the Filesystem

Up till now, we have discussed building a toolchain, bootloader (U-boot), Linux kernel, and now we will talk about the filesystem. We will be following the same sequence we used to build the prior elements.
For the first step, instead of downloading some source code through Github, we will be installing a tool called BusyBox that will aid us in generating a root file system.

BusyBox is open-source and licensed under the GPL. BusyBox is a collection of core UNIX utilities packaged as a single binary. This makes it ideal for resource-constrained environments, which is just a fancy way to describe an embedded system. The complete distribution has almost 400 of the most common commands.
BusyBox input and output can be used in a standard manner across different distributions and versions of Linux. This helps keep the system configuration files compatible between different versions of Linux and allows easy updating of the system without having to learn new commands or make modifications. In addition, it can also help automate tedious tasks so less time is spent maintaining the system, saving valuable resources and time.

BusyBox provides the everyday convenience commands that often feel like they're part of your shell. Although userland tools like ls and cat are ubiquitous, they actually reside in a separate utility package that's independent of your shell. Many Linux distributions deliver these commands via GNU's coreutils but others ship BusyBox instead.

12.3.6 Building the Application

Embedded Linux can run a wide variety of programming languages, it can run Python, C/C++, Rust, etc. It all comes down to what application we want to develop.

Let's for example look at our case, where we had used the traditional method, we would be running our code on a cross compiler to produce an executable that we'll run on our target device. Of course, the cross compiler is going to differ according to the target device, as for Raspberry Pi, it will be an arm cross compiler.

12.4 Using Yocto to build The Image

After exploring the traditional approach to customizing a Linux image, we will delve into the details of The Yocto Project, an alternative method of customizing a Linux image that we employed in our project.

Figure 12.1

The Yocto Project is a comprehensive open-source embedded Linux build system that enables the rapid development of custom embedded Linux distributions and related components. It provides an intuitive user interface to easily configure, create, and deploy these distributions to multiple target platforms. With its wide array of tools and libraries, it simplifies the implementation of complex multi-level layers for product specific customization and board support packages (BSPs). Furthermore, its adaptability helps manufacturers reduce time-to-market while increasing efficiency and extensibility in products with integrated Yocto components. By using packages such as Poky and OpenEmbedded Core (OE-Core), developers have access to pre-configured as well as customizable software stacks that can be used across multiple architectures including ARM, PowerPC, MIPS, x86 or x86_64 processors. That makes Yocto one of the most accepted open source embedded build systems around.

The Yocto Project provides a compliant framework for creating customized Linux distributions for embedded and IoT devices. Its components include the OpenEmbedded-Core layer, which is a set of core recipes and classes to support complex image customization operations; the Poky layer, which offers technology to construct any type of Linux distribution; the BitBake build system which is responsible for the integration process of all other layers; Configurations files, used to interact effectively with multiple host systems to ensure easy cross-compilation; and Templates, providing helpful utilities and hooks enabling users to create very specific customizations. All said components allow the Yocto Project to offer developers advanced flexibility in building custom solutions for their applications in the embedded market. In the following paragraphs, we will delve deeper into the structure of the Yocto Project and all of its components.

Using reference templates, layers, build tools and machine configurations, it facilitates software integration activities across different hardware architectures. The project utilizes a shared metadata that allows developers to easily manage and customize the more than 3,000 electronic components supported by its ecosystem. Uniquely among Linux embedded platforms, Yocto provides developers with unparalleled flexibility across I/O devices, peripherals, and boards. In addition to established drivers and frameworks such as Linux kernel modules, multimedia codecs and hardware adaptation capabilities relying on board support packages features from 3rd party vendors; all aspects of the development environment can be tailored or extended according to specific requirements.
The following figure illustrates the Yocto Project Environment.

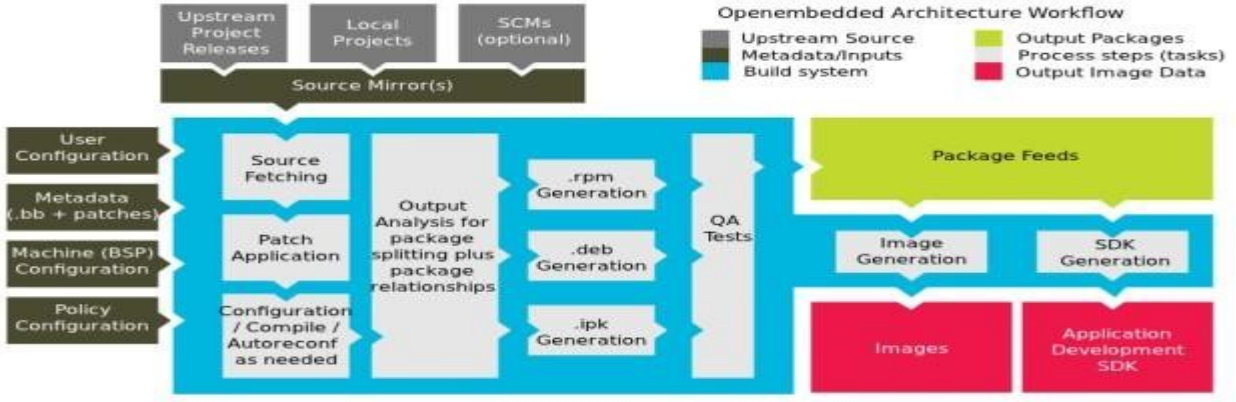


Figure 12.2

Now that we have untangled what the Yocto Project is, let's demonstrate the components and tools it's built upon.

12.4.1 Poky

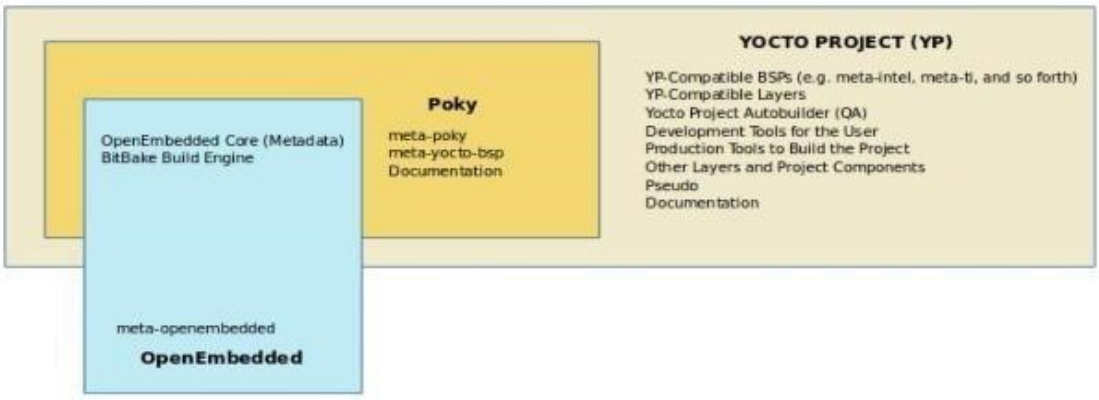


Figure 12.3

It is the reference distribution or the reference OS kit of the Yocto Project. It provides a standard set of metadata and tooling that serves as a foundation layer for other distributions and products in the project ecosystem. Poky includes routines to build Embedded Linux systems with cross compiling support, as well as applications tailored to embedded environments such as BitBake and OE Core. While Poky is a "complete" distribution specification and is tested and put through QA, we cannot use it as a product "out of the box" in its current form.

12.4.2 Board Support Package (BSP)

It is a collection of packages and recipes that allow operating systems to be customized for embedded systems. The BSP package includes all necessary files for building a customized operating system for an embedded board, such as board-specific device support, graphics drivers, startup scripts, bootloaders, and kernel modules. Poky supports multiple BSPs by default like x86, Beaglebone, and Freescale platform.

12.4.3 Metadata

It is basically the layers about a specific hardware or a software package. It abstracts away all the cross-compilation complexity and makes it easy for users to simply identify what packages are needed. This reduces development time by

streamlining component selection, enabling faster delivery of products built around the Yocto Project technology stack. Metadata can be:

Configurations Files (.conf) like Machine type, packages to be installed, etc.

Recipes are the most common form of metadata. A recipe contains a list of settings and tasks (i.e. instructions) for building packages that are then used to build the binary image. A recipe describes where you get source code and which patches to apply. Recipes describe dependencies for libraries or for other recipes as well as configuration and compilation options.

Related recipes are consolidated into a layer. Recipes (.bb or .bbappend) (BitBake) files are makefile-like recipes used within the Yocto Project to build packages. Each recipe file contains all the important information to build a component, including what source code is needed for the component, how it should be configured, and which dependencies must be satisfied before building it. Furthermore, bb files also contain instructions on how to package and install a component once it has been built. Additionally, since source code typically used in Yocto projects is often very complicated or infrequently used, bb files help developers focus on only the specific parts that really matter when dealing with them.
Classes (.bbclass) files enable developers to create their own recipes using classes. Class files provide a means to share common code and configuration options between packages. This allows developers to reduce the number of lines of code they need to write, while also avoiding the duplication of code, which can lead to error-prone applications. Additionally, with class files, developers can quickly access settings and variables shared across multiple packages without having to look up each setting separately.

Includes (.inc) files can be used to configure the kernel, add software packages, or modify system settings. Additionally, they can also store recipes in order to define how a specific package would be built during image generation.

12.4.4 Open-Embedded Core (OE-Core)

It provides developers with the necessary tools to build their own customized Linux distributions for embedded targets. OE-Core takes care of dependency management and cross compilation support. This ensures that platforms such as ARM can run sophisticated applications while still utilizing minimal system requirements. With its powerful yet easy to use feature set, Yocto's OpenEmbedded Core makes using Linux on embedded devices easier than ever.

12.4.5 BitBake

It is a powerful tool used by the Yocto Project to support cross-platform, embedded Linux development. At its core, it takes project metadata and recipes as inputs, executes user commands and tasks, and outputs binary packages. Furthermore, it is a sophisticated build system that can handle dynamic runtime dependencies between tasks during execution via an internal dependency engine.

12.4.6 Layer

A collection of related recipes. Layers allow us to consolidate related metadata to customize our build. Layers also isolate information used when building for multiple architectures. Moreover, Layers are hierarchical in their ability to override previous specifications. We can include any number of available layers from the Yocto Project and customize the build by adding your layers after them. We can search the Layer Index for layers used within Yocto Project.

12.4.7 Packages

In the context of the Yocto Project, this term refers to a recipe's packaged output produced by BitBake (i.e., a "baked recipe"). A package is generally the compiled binaries produced from the recipe's sources. We "bake" something by running it through BitBake.

12.4.8 QEMU

It is an open-source virtual machine developed by the Linux Foundation in cooperation with Intel and is well-suited for use with the Yocto Project. By utilizing QEMU in an embedded system, it allows systems engineers a safe environment to conduct experiments without interfering with actual hardware or its real-time

behavior. Simply, it acts as a simulator to run an image to test it and to know whether it is working or not and to know if there are any bugs. [74]

To be able to use Yocto Project, Poky is downloaded to the Host Machine and its file structure contains:

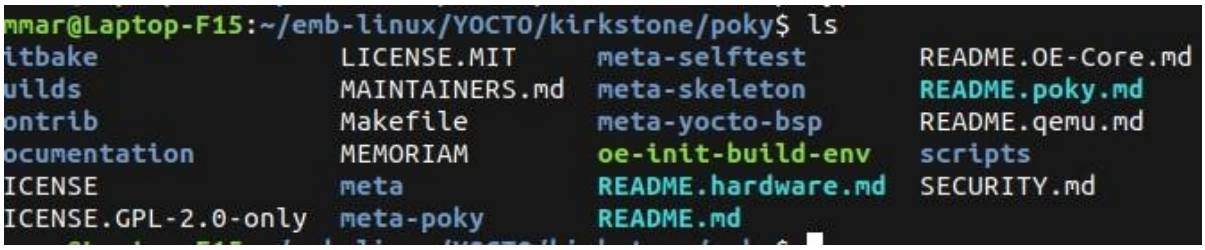


Figure 12.4

- **Meta**

  It is the Open-Embedded meta data.
- **Meta-poky**

  It is the meta data of Yocto Project.
- **Meta-self test**

  Meta data used while building and testing the image, but it is not included in the image.
- **Meta-skeleton**

  A template used as a layer example.
- **Meta-yocto-bsp**

  It contains the BSPs supported by poky like x86, BeagleBone and FreeScale.

CHAPTER 13:
Main System (GUI)

13.1 What is GUI?

A Graphical User Interface (GUI) is a form of user interface that allows users to interact with electronic devices through graphical icons and visual indicators such as secondary notation. In many applications, GUIs are used instead of text-based UIs, which

are based on typed command labels or text navigation. GUIs were introduced in reaction to the perceived steep learning curve of command-line interfaces (CLIs), which require commands to be typed on a computer keyboard.

13.2 Why do we need GUI in cars?

A Graphical User Interface (GUI) in cars is essential for enhancing driver and passenger interaction with the vehicle's systems in a safe, intuitive, and efficient manner. Modern vehicles are equipped with a wide range of features—such as navigation, entertainment, climate control, and advanced driver assistance systems (ADAS)—that require clear and accessible user controls.

A well-designed GUI presents this information visually, allowing users to easily understand and operate various functions without distraction. By minimizing cognitive load and reducing the need for manual inputs, GUIs contribute to improved safety, user experience, and overall vehicle functionality, making them a critical component of today's automotive design.

13.3 What is QML?

The Modern, Declarative Approach QML (Qt Modeling Language) is a declarative language developed by the Qt framework, specifically designed for building modern, fluid, and highly dynamic graphical user interfaces (GUIs). It allows developers to design user interfaces in a readable and compact syntax, separating the UI design from the backend logic written in C++ or JavaScript. QML is especially effective for applications requiring a rich, animated, and responsive user experience, making it an ideal choice for embedded systems, mobile applications, and in-vehicle infotainment systems.

One of the key advantages of QML is its ability to simplify UI development through a component-based architecture, enabling developers and designers to reuse elements efficiently and iterate quickly. It supports seamless integration with C++ for performance-critical tasks, offering both visual richness and computational efficiency. Additionally, QML's real-time responsiveness and support for animations, transitions, and touch gestures enhance the interactivity and appeal of user interfaces. Its tight integration with the Qt framework ensures cross-platform compatibility, making it a

powerful tool for developing consistent UIs across different devices and operating systems. Overall, QML significantly reduces development time and complexity while delivering hig

Pros of QML

1. Ease of Use: QML's declarative syntax is straightforward, allowing for faster prototyping and UI iteration.
2. Powerful Animation Capabilities: QML excels in creating animations, transitions, and interactive elements, making it perfect for dynamic UIs.
3. Mobile and Touch-Friendly: QML is optimized for mobile devices and offers touch support, making it the preferred choice for cross-platform mobile apps.
4. Rapid Development Cycle: QML allows developers and designers to iterate quickly, ideal for modern UIs that require frequent changes.
5. Integration with JavaScript: JavaScript support in QML makes it easier to manage UI logic without extensive C++ code.

Cons of QML

1. Learning Curve for C++ Developers: Developers with a strong C++ background may find QML's JavaScript syntax challenging at first.
2. Less Control Over Low-Level Performance: For applications with strict performance requirements, QML may require C++ integration, adding complexity.
3. Limited Debugging Tools: QML debugging can be less intuitive, though tools are improving in newer Qt versions.
4. Native Look and Feel Challenges: Unlike QtWidgets, QML doesn't provide native UI elements by default, so it may not always look or behave like a traditional desktop app.

When to Use QML? QML is ideal for: Mobile Applications where touch and animation are critical. UI-Driven Applications that prioritize user experience and visual appeal, like media or entertainment apps. Prototyping and Rapid UI Development due to its flexibility and ease of iteration.

13.4 System Requirements

The system is composed of both hardware and software components: Software: Operating System: Linux OS. Hardware: Raspberry Pi 4: Serves as the processing unit.

13.5 System Implementation

1- Main Window

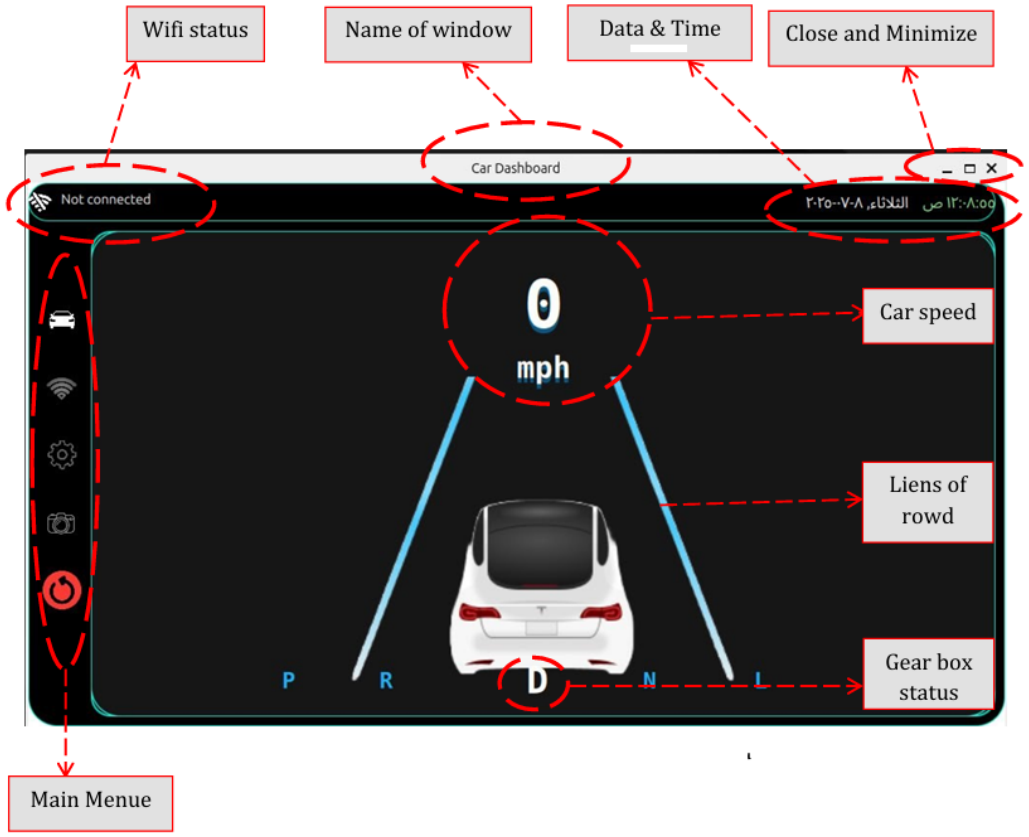


Figure 13.1

1. WiFi Status

The WiFi status indicator displays the current network connectivity of the vehicle's system. It informs the user whether the system is connected to a wireless network, enabling features like internet access, software updates, cloud services, and online media. In the shown example, it displays "Not connected", which alerts the driver to connect the system to a network if necessary for full functionality.

2. Name of Window

The Name of Window, labeled as "Car Dashboard," serves as a clear identifier for the active application or module running on the car's infotainment system. It provides context to the user, ensuring they know which system interface they are currently interacting with. This is especially important in multitasking

environments where multiple screens or functions may be accessed simultaneously.

3. Date & Time

The Date and Time display provides real-time temporal information, helping drivers stay aware of the current day and time while driving. This information can be useful for navigation, scheduling, or logging purposes. Displaying it prominently on the dashboard enhances the utility of the interface without requiring additional user interaction.

4. Car Speed

The Car Speed display shows the current speed of the vehicle in miles per hour (mph). Positioned centrally for maximum visibility, this is a core safety and performance metric, allowing the driver to monitor speed at a glance. This real time feedback helps maintain safe driving and avoid exceeding speed limits.

5. Lines of Road

The Lines of Road visualization mimics a forward camera or ADAS-based lane keeping system. It assists the driver in maintaining lane discipline and provides a virtual representation of the car's positioning relative to lane markers. This feature is particularly valuable for enhancing road awareness and supporting semi autonomous driving modes.

6. Gear Box Status

The Gear Box Status section displays the current gear selected by the vehicle (P, R, N, D). This is crucial for informing the driver of the vehicle’s operational state—whether it's in park, reverse, neutral, or drive. Such feedback helps avoid incorrect gear usage and supports safe vehicle control, especially during low-speed maneuvers or parking.

7. Main Menu

The Main Menu located on the left sidebar provides access to various subsystems of the car’s dashboard, such as navigation, settings, camera views, system status, or multimedia controls. This menu serves as a navigation hub, ensuring users can quickly switch between modules through intuitive icon-based buttons without interrupting their driving experience.

8. Close & Minimize

This option to close and minimize the window.

2- Wi-Fi Window

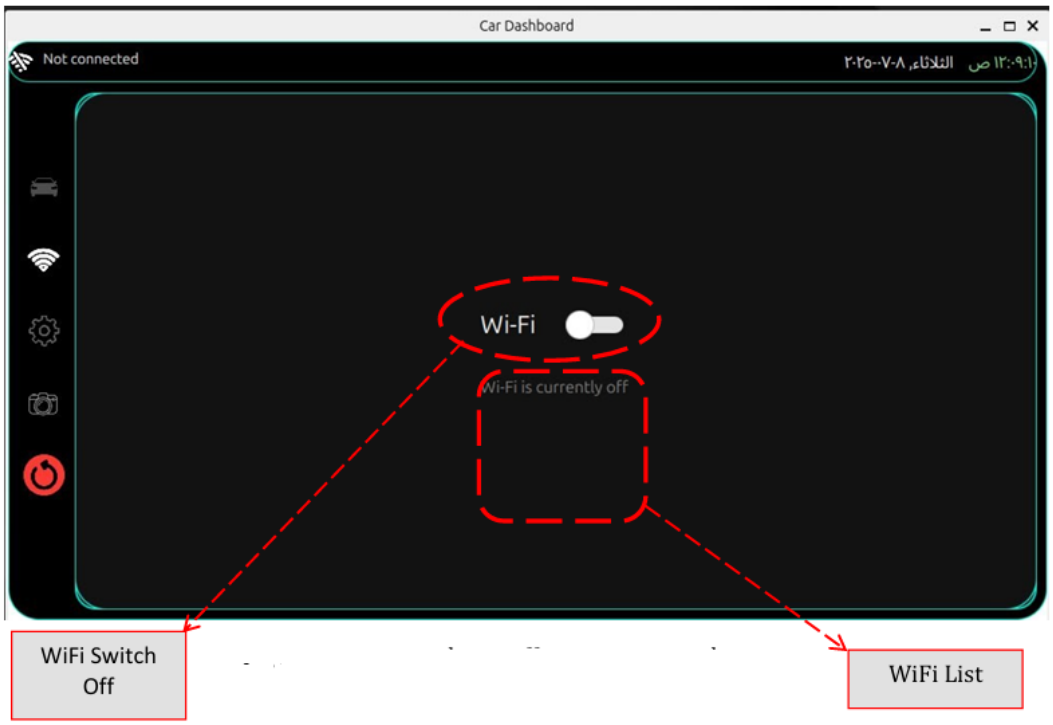


Figure 13.2

1. Wi-Fi Switch

This toggle button allows the user to enable or disable the Wi-Fi module in the car dashboard. When the switch is in the off position, the system will display a message such as "Wi-Fi is currently off" and will not attempt to scan or connect to any available networks. Toggling it to the on position activates the Wi-Fi interface, allowing the dashboard to search for nearby Wi-Fi networks and connect to them for data services or updates.

2. Wi-Fi Status

Message Located below the Wi-Fi switch, this text element dynamically shows the current state of the Wi-Fi system. In the provided image, it reads "Wi-Fi is currently off", which informs the user that no Wi-Fi activity is ongoing. When the Wi-Fi is turned on, this message would update to reflect the scanning status, connection attempts, or indicate successful connection along with the network name.

3. Wi-Fi List

This section, which appears blank in the current image, is designed to display the list of available Wi-Fi networks when the Wi-Fi switch is turned on. Each network would typically be shown with details such as the SSID (network name), signal strength, and whether the network is secured (requires a password). Users can tap on a network from this list to initiate a connection.

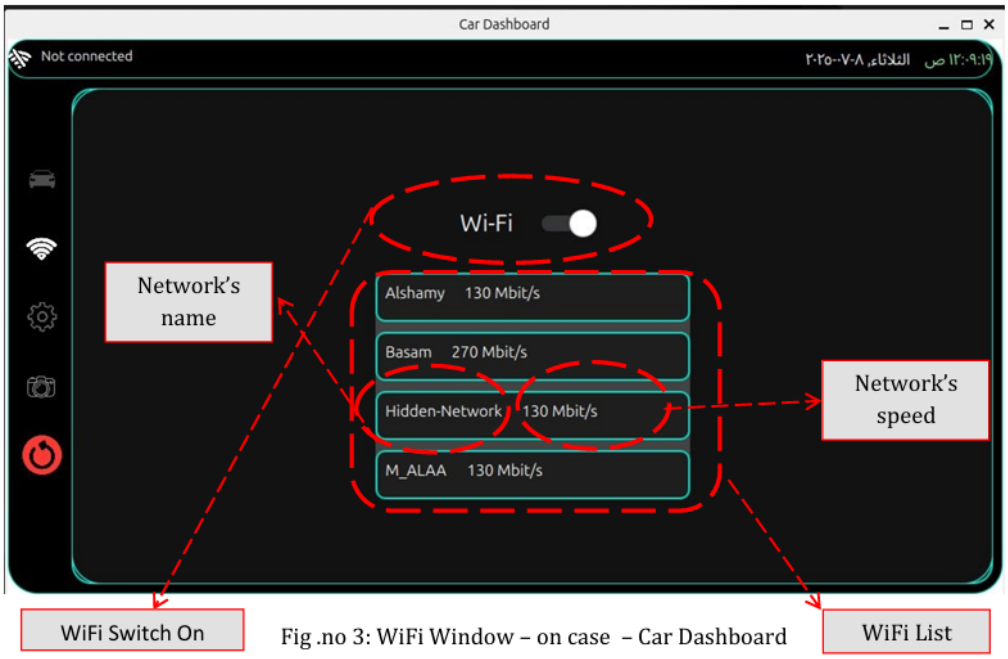


Fig .no 3: WiFi Window – on case – Car Dashboard

Figure 13.3

1. WiFi Switch (On/Off Toggle)

The WiFi Switch is a crucial control element that allows the user to enable or disable the car's wireless connectivity. When set to "On," the system scans for available WiFi networks and displays them for connection. Turning this switch off disables all WiFi-related operations to conserve power or for privacy/security purposes. This toggle ensures that the driver or user has full control over the vehicle's network accessibility.

2. WiFi List

The WiFi List section presents all detected WiFi networks within range of the car. Each entry includes the network's name (SSID) and signal quality or speed, allowing the user to select the most suitable connection. This list is dynamically updated as the vehicle moves through different areas, ensuring real-time visibility of connectivity options, which is vital for navigation, streaming, and connected services.

3. Network's Name (SSID)

Each WiFi entry includes the Network's Name, commonly known as the SSID (Service Set Identifier). This label helps users identify familiar networks, such as home routers, public hotspots, or mobile tethering sources. Displaying the SSID allows for quick recognition and informed connection choices, especially when multiple networks are available.

4. Network's Speed

The Network's Speed value shows the theoretical maximum connection speed in Mbps (e.g., 130 Mbps, 270 Mbps). This detail assists users in selecting the fastest and most stable connection, which is especially important for tasks like streaming, OTA updates, or using cloud-based applications in the car. It provides a quick performance indicator to ensure optimal user experience.

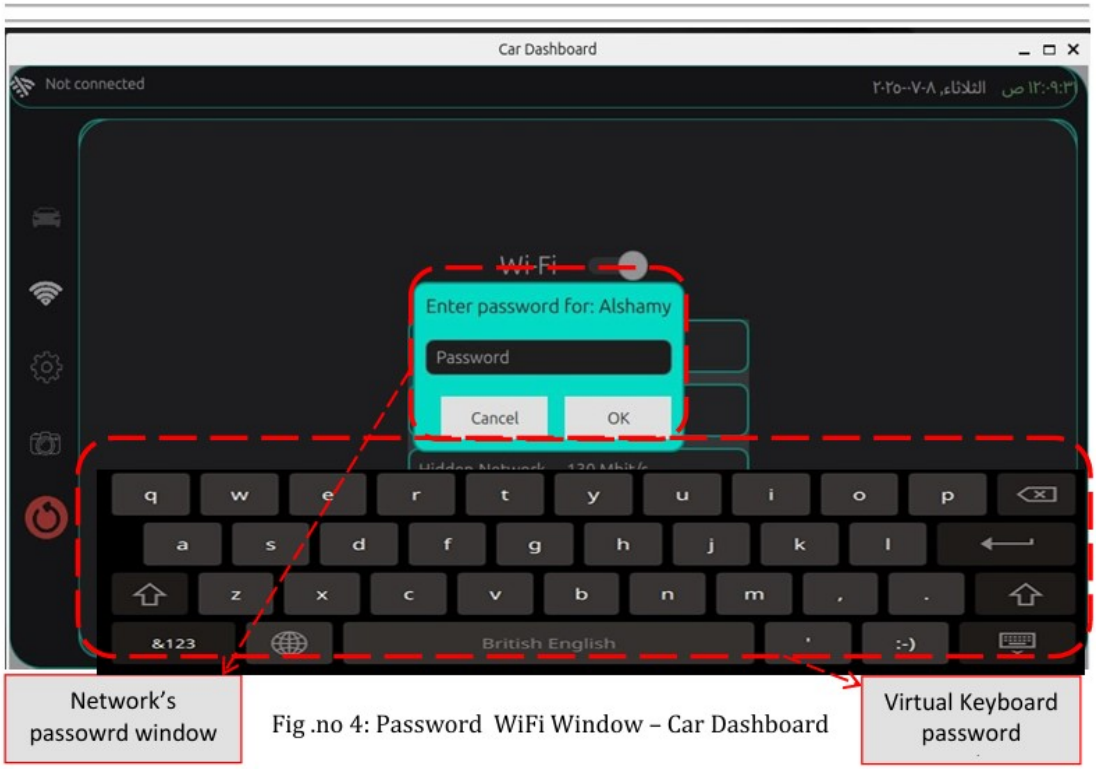


Fig .no 4: Password WiFi Window – Car Dashboard

Figure 13.4

1. Network Selection

This tab is triggered when a user taps on one of the listed Wi-Fi networks from the available Wi-Fi list (as shown in the previous image). Upon selection of a secured network, the system prompts the user to enter a password, ensuring secure connection and preventing unauthorized access. In this case, the network "Alshamy" has been selected.

2. Password Entry Field

This is the main input field where users type the Wi-Fi password. It is labeled "Password" and accepts alphanumeric characters. Typically, this field will mask the input to prevent shoulder-surfing or unauthorized viewing of the password. This ensures security while entering sensitive credentials.

3.OK Button

Once the user has entered the password, pressing the "OK" button initiates the connection process to the selected Wi-Fi network. If the credentials are valid, the system will connect to the network and may update the status to "Connected" in the status bar. If incorrect, it might return a prompt or error notification.

4. Cancel Button

The "Cancel" button allows the user to abort the password entry process and return to the main Wi-Fi list without attempting a connection. This is useful if the user selects a network by mistake or decides not to proceed with connecting.

5. Virtual Keyboard

The Virtual Keyboard is an essential user interface element integrated into the car dashboard system, designed to facilitate text input without the need for physical hardware. It appears automatically when input fields require user interaction—such as entering WiFi passwords, searching for media, or setting navigation destinations. This on-screen keyboard supports touch input, allowing drivers or passengers to enter information easily and intuitively. Its layout mimics that of a standard keyboard and may include features such as auto-correction, multi language support, and special character input. The inclusion of a virtual keyboard enhances system accessibility, reduces hardware clutter, and supports a seamless user experience, especially in embedded automotive environments where space and safety are critical.

3- Control Window

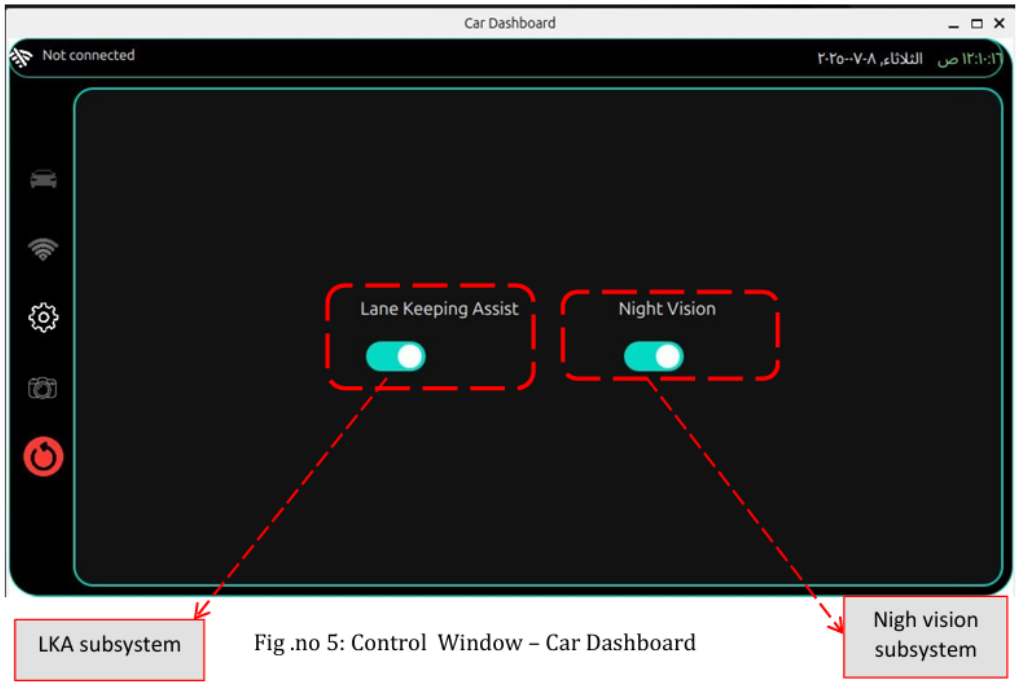


Fig .no 5: Control Window – Car Dashboard

Figure 13.5

1. Lane Keeping Assist Toggle

This switch enables or disables the Lane Departure Warning (LDW). When activated (as shown in the image), the system uses input from cameras and sensors to detect lane markings on the road and provides corrective steering inputs to keep the vehicle centered. This feature is especially useful during long highway drives to reduce driver fatigue and improve safety by minimizing unintentional lane departure.

2. Night Vision Toggle

The Night Vision toggle controls an advanced driver-assistance system that enhances visibility during nighttime or low-light conditions. When turned on, this system uses infrared cameras or sensors to detect obstacles, pedestrians, or animals beyond the range of standard headlights, projecting the enhanced view onto the dashboard or HUD. This feature is crucial for preventing accidents and increasing situational awareness in dark environments.

4- Camera Window

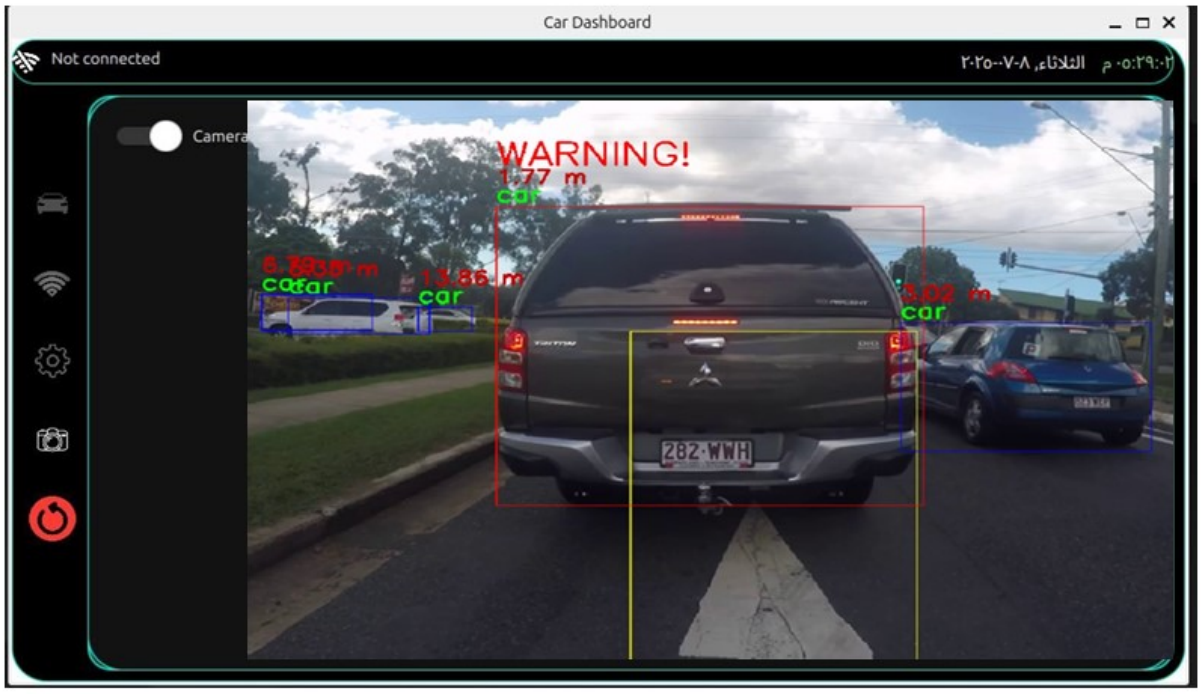


Figure 13.6

1. Camera Toggle Switch

At the top-left corner of the feed window is a toggle switch labeled “Camera.” This button is used to activate or deactivate the camera feed. If turned off, the visual feed and detection processes are paused to conserve system resources or during maintenance.

2. Camera Live Feed Panel

The central part of the screen is dedicated to displaying the live camera feed from the vehicle’s front-facing camera. This panel plays a critical role in visual monitoring and object detection. It visually assists the driver by showing exactly what the car sees in real-time. The use of bounding boxes and class labels over detected objects makes it easier to understand the surrounding environment dynamically.

3. Object Detection and Distance Estimation

Over the video feed, the system uses object detection algorithms to identify other vehicles in the vicinity. These objects are enclosed in colored bounding boxes and labeled with both the object type (e.g., "car") and its distance from the user's car (e.g., "0.77 m"). This functionality enhances situational awareness and is vital for collision avoidance, autonomous driving, or driver-assist systems.

5- Applications Window

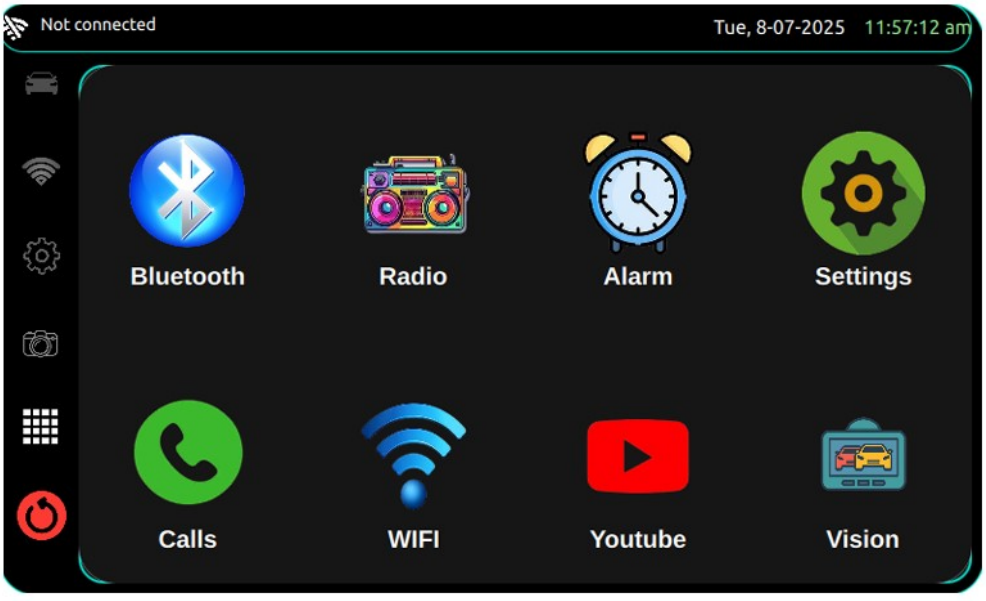


Figure 13.7

1. Bluetooth

The Bluetooth feature on the car dashboard enables wireless connectivity between the vehicle and external devices such as smartphones, tablets, and audio systems. This function allows for hands-free calling, media streaming, and synchronization of contacts and call logs. In modern automotive systems, Bluetooth plays a crucial role in enhancing driver convenience and safety by minimizing distractions.

2. Radio

The Radio tab provides access to the car's built-in FM/AM radio system. It allows users to browse, select, and store their favorite stations while displaying current frequency information. Designed for easy navigation and control, this feature ensures entertainment and access to live news or traffic updates during travel. The colorful icon enhances user engagement and highlights its multimedia role in the system.

3. Alarm

The Alarm function acts as a scheduling and notification tool within the dashboard. Drivers can set alarms for various purposes such as wake-up reminders, maintenance checks, or time-sensitive alerts. Its clock-based interface makes it intuitive and familiar. This feature supports time management and can be particularly helpful for long-distance travelers or fleet drivers.

4. Settings

The Settings menu provides access to system configurations, allowing users to customize various parameters of the car dashboard. These include display brightness, sound volume, language options, date/time settings, and more. This tab serves as the control center for tailoring the system’s behavior to individual preferences, enhancing user experience and operational efficiency.

5. Calls

The Calls tab allows drivers to make and receive phone calls directly through the vehicle's interface via a paired device. It integrates with Bluetooth to provide contact access, call history, and keypad dialing. By offering a hands-free calling environment, this feature supports safer driving while maintaining communication with minimal distraction.

6. WiFi

The WiFi feature enables the car to connect to wireless internet networks. This may include connecting to a mobile hotspot, home WiFi, or public access points. It plays a vital role in enabling online services such as navigation updates, streaming, and cloud connectivity. Secure and stable WiFi access enhances the car's smart capabilities and data-driven features.

7. YouTube

The YouTube application allows users to access and stream videos directly on the car dashboard. Though generally used while the vehicle is stationary, it provides entertainment for passengers and is especially valuable during long trips or when parked. Parental controls or driver lockouts may be implemented to ensure safety and compliance with driving regulations.

8. Vision

The Vision feature likely refers to an advanced driver assistance system (ADAS) or a camera-based monitoring tool. It could include functionality like rear-view camera feeds, lane detection, obstacle alerts, or night vision support. This component significantly boosts safety by providing real-time visual feedback, aiding in navigation, parking, and overall situational awareness.

6- Restart Button

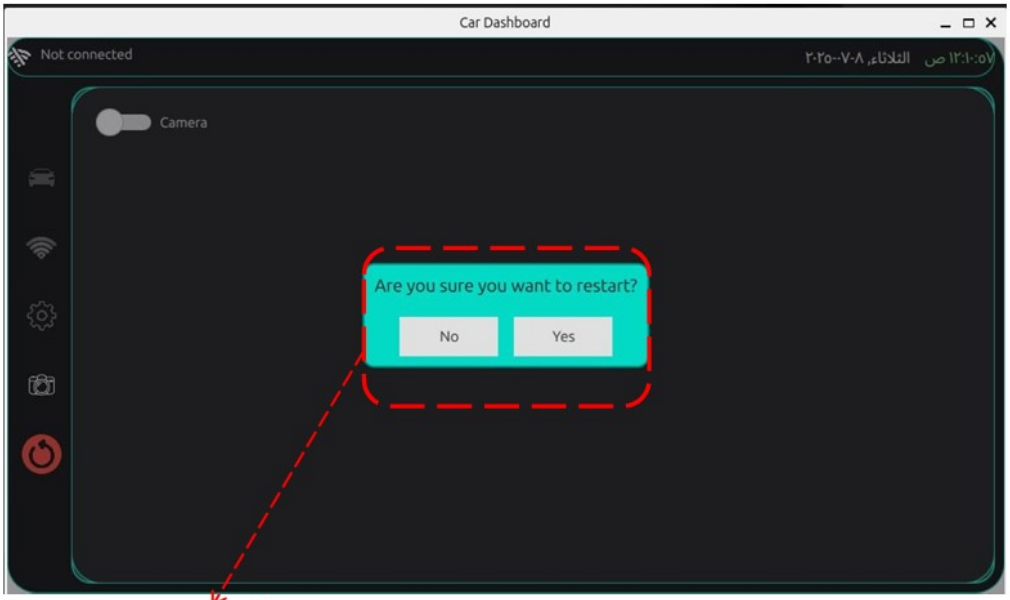


Fig .no 8: Restart Window – Car Dashboard

Figure 13.8

1. Restart Confirmation Dialog Box

At the center of the screen is a modal dialog box prompting the user with the message: “Are you sure you want to restart?” This is a confirmation step that prevents accidental system restarts. It's a key safety feature, ensuring that important functions like live video processing or connectivity are not interrupted unintentionally. The dialog includes two options: Yes: Confirms the restart, initiating a system reboot or application reset. No: Cancels the action, closing the dialog box and returning the user to the main dashboard.

CHAPTER 14 :
Embedded Linux Development using Yocto Project

14.1 Preparing the build Host system

To use Yocto Project on the host machine, Make sure your Build Host meets the following Requirements:

- 50 Gigabytes of free disk space.
- Runs a supported Linux distribution (i.e. recent releases of Fedora, openSUSE, CentOS,
- Debian, or Ubuntu).
- In our project we are using Ubuntu 22.04
  - Git 1.8.3.1 or greater.
  - tar 1.28 or greater.
  - Python 3.5.0 or greater.
  - gcc 5.0 or greater.

14.2 Downloading the Poky Source Code

Once you complete the setup instructions for your machine, you need to get a copy of the Poky repository on your build host. Use the following commands to clone the Poky repository:

```
git clone git://git.yoctoproject.org/poky
```

After the download process is complete, we should have the following contents inside the poky directory:

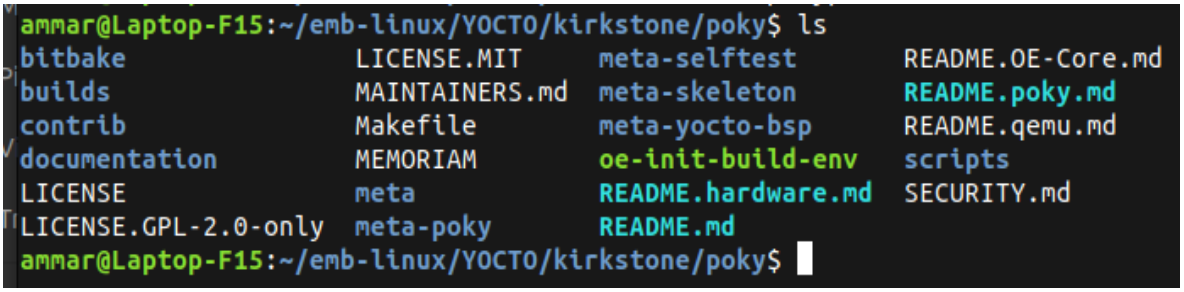


Figure 14.1

The last step in the installation process is to choose the release, which we will be working on, through their corresponding code name, each representing a branch from the Poky repository with its own support lifetime. In our project we have chosen **kirkstone** release:

14.3 Creating active Build directory

A build directory is where all your Yocto builds reside. What this allows you to do is to compartmentalize the builds cleanly but at the same time re-use as many resources as possible.

In the build directory resides the machine configuration, specific variables that your image/machine needs, additional recipes you need added to your image, etc.

At the same time, the reusable components like common recipes, source code, tarballs, build state cache, etc. all reside in the **poky** folder.

This allows you to simply create a new build directory for your other machines or projects but still reuse existing components already downloaded, created and/or built.

To create a build directory, you simply need to source a script called oe-init-build-env that

will create the build environment (paths, shortcuts, variables, etc.).

```
source oe-init-build-env
```

```
ammar@Laptop-F15:~/emb-linux/YOCTO/kirkstone/poky$ source oe-init-build-env
You had no conf/local.conf file. This configuration file has therefore been
created for you from /home/ammar/emb-linux/YOCTO/kirkstone/poky/meta-poky/conf/local.conf.sample
You may wish to edit it to, for example, select a different MACHINE (target
hardware). See conf/local.conf for more information as common configuration
options are commented.

You had no conf/bblayers.conf file. This configuration file has therefore been
created for you from /home/ammar/emb-linux/YOCTO/kirkstone/poky/meta-poky/conf/bblayers.conf.sample
To add additional metadata layers into your configuration please add entries
to conf/bblayers.conf.

The Yocto Project has extensive documentation about OE including a reference
manual which can be found at:
    https://docs.yoctoproject.org

For more information about OpenEmbedded see the website:
    https://www.openembedded.org/


### Shell environment set up for builds. ###

You can now run 'bitbake <target>'

Common targets are:
    core-image-minimal
    core-image-full-cmdline
    core-image-sato
    core-image-weston
    meta-toolchain
    meta-ide-support

You can also run generated qemu images with a command like 'runqemu qemux86'

Other commonly useful commands are:
 - 'devtool' and 'recipetool' handle common recipe tasks
 - 'bitbake-layers' handles common layer tasks
 - 'oe-pkgdata-util' handles common target package tasks
```

Figure 14.2

sourcing the build environment

Looking inside the build directory, we have the following directories:

```
ammar@Laptop-F15:~/emb-linux/YOCTO/kirkstone/poky/build$ ls ../../layers/meta-mender-community/build/
bitbake-cookerdaemon.log  bitbake.lock  bitbake.sock  cache  conf  downloads  hashserve.sock  sstate-cache  tmp
```

Figure 14.3

1- conf: contains the following files:

+ **bblayers.conf:** BitBake uses the build/conf/bblayers.conf file to list the layers considered in the build environment.
+ local.conf: The build/conf/local.conf file is the placeholder for the local configurations like machine type, package classes, configuration version and many more:

a. Set the machine which we will be working on (raspberrypi3 in our case). There is a various selection of emulated machines available beside the QEMU emulator.

b. Set where to place downloads directory: As during the first build the system will download many different source code tarballs from various upstream projects. This can take a while, particularly if the network connection is slow. These are all stored in DL_DIR. When wiping and rebuilding we can preserve this directory to speed up this part of subsequent builds. This directory is safe to share between multiple builds on the same machine too.

c. Set where to place the shared-state files directory: Since BitBake has the capability to
accelerate builds based on previously built output. This is done using "shared state" files which can be thought of as cache objects and this option determines where those files are placed. We can wipe out TMPDIR leaving this directory intact and the build would regenerate from these files if no changes were made to the configuration. If changes were made to the configuration, only shared state files where the state was still valid would be used (done using checksums).

d. Set where to place the build output (tmp file): This option specifies where the bulk of the building work should be done and where BitBake should place its temporary files and output. Keep in mind that this includes the extraction and compilation of many applications and the toolchain which can use Gigabytes of hard disk space.

e. Set the distribution settings: The distribution setting controls which policy settings are used as defaults. The default value (Poky) is fine for general Yocto project use, at least initially. Ultimately when creating custom policy, people will likely end up sub classing these defaults.

f. Enable/Disable package management configurations.

g. Add extra packages to the generated images through EXTRA_IMAGE_FEATURES.

h. QEMU configurations in case we're using QEMU.

i. Optimization options for maximum build time. In addition to many other features, these were the main ones.

```
##################################################################
# raspberrypi
##################################################################
LICENSE_FLAGS_ACCEPTED += "commercial synaptics-killswitch"
RPI_USE_U_BOOT = "1"
ENABLE_UART = "1"
# MENDER_BOOT_PART_SIZE_MB is already set above; duplicate removed to avoid confusi
IMAGE_INSTALL:append = " kernel-image kernel-devicetree"
IMAGE_FSTYPES:remove = " rpi-sdimg"
MACHINE ??= "raspberrypi4-64"
DISTRO ??= "poky"
# Enables camera support
VIDEO_CAMERA = "1"
# Enables 2-channel CAN FD controller
KERNEL_DEVICETREE:append = " overlays/mcp251xfd.dtbo"
# # Enables I2C subsystem
# ENABLE_I2C = "1"
# # Auto-loads I2C drivers
# KERNEL_MODULE_AUTOLOAD:rpi += "i2c-dev i2c-bcm2708"

# Enable X11
DISTRO_FEATURES:append = " x11 opengl"

# Disable systemd-networkd in favor of NetworkManager
DISTRO_FEATURES:append = " networkmanager"
VIRTUAL-RUNTIME_net_manager = "networkmanager"

# Enable X11
X11_PKGS = "\
    packagegroup-core-x11 \
    xserver-xorg \
    xf86-video-fbdev \
    xterm \
    matchbox-terminal \
    packagegroup-core-x11 \
    packagegroup-core-x11-base \
    matchbox-session \
    matchbox-terminal \
    dbus-x11 \
"
```

Figure 14.4

2- **downloads:** it contains the fetched files and packages that will be installed to the custom image.

3- **sstate-cache:** It contains the shared state cache. It will be used later to act as a base for multiple builds. This directory can be moved to another location and specify in the build configuration of the image where it is through the SSTATE_DIR variable.

4- **tmp:** The OpenEmbedded build system creates and uses this directory for all build system's output. BitBake creates this directory if it does not exist. As a last resort, to clean up a build and start it from scratch (other than the downloads), we can

remove everything in the tmp directory or get rid of the directory completely. Note that if we do so, we should also completely remove the state-cache. tmp has the following directories:

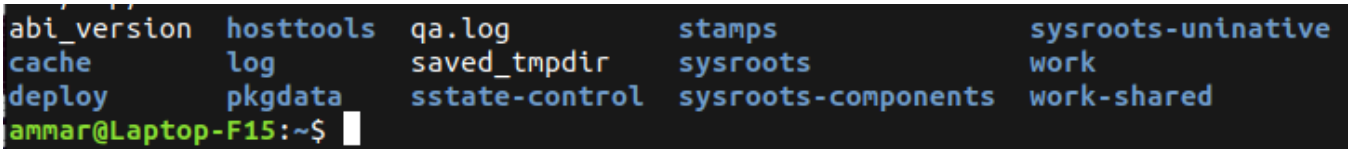

abi_version hosttools qa.log stamps sysroots-uninative
cache log saved_tmpdir sysroots work
deploy pkgdata sstate-control sysroots-components work-shared
ammar@Laptop-F15:~$

Figure 14.5

- **buildstats:** stores the build statistics.

- **cache:** after bitbake parses the metadata (recipes and configuration files), it caches the

results tmp/cache to speed up future builds. During subsequent builds, Bitbake checks each recipe (together with, for example, any files included or appended to it) to see if they have been modified. Changes can be detected, for example, through file modification time (mtime) changes and hashing of file contents. If no changes to the file are detected, then the parsed result stored in the cache is reused. If the file has changed, it is reparsed.

- **deploy:** This directory contains any "end result" output from the OpenEmbedded build process. The DEPLOY_DIR variable points to this directory.

- **deb:** This directory receives any .deb packages produced by the build process. The packages are sorted into feeds for different architecture types.

- **rpm:** This directory receives any .rpm packages produced by the build process. The packages are sorted into feeds for different architecture types.

- **ipk:** This directory receives .ipk packages produced by the build process.
- licenses: This directory receives package licensing information. For example, the directory contains sub-directories for bash, busybox, and glibc (among others) that in turn contain appropriate COPYING license files with other licensing information.

- **Images:** This directory is populated with the basic output objects of the build (think of them as the "generated artifacts" of the build process), including things like the boot loader image, kernel, root filesystem, and more. If we want to flash the

resulting image from a build onto a device, look here for the necessary components. Note that when deleting files in this directory.
We can safely delete old images from this directory (e.g. core-image-*). However, the kernel (*zImage*, *uImage*, etc.), bootloader, and other supplementary files might be deployedhere prior to building an image. Because these files are not directly produced from the image,
if we delete them they will not be automatically re-created when we build the image again.

- **sdk:** The OpenEmbedded build system creates this directory to hold toolchain installer scripts which, when executed, install the sysroot that matches your target hardware.
- **sstate-control:** The OpenEmbedded build system uses this directory for the shared state manifest files. The shared state code uses these files to record the files installed by each sstate task so that the files can be removed when cleaning the recipe or when a newer version is about to be installed. The build system also uses the manifests to detect and produce a warning when files from one task are overwriting those from another.

- **sysroots-components:** This directory is the location of the sysroot contents that the task do_prepare_recipe_sysroot links or copies into the recipe-specific sysroot for each recipe listed in DEPENDS. Population of this directory is handled through shared state, while the path is specified by the COMPONENTS_DIR variable. Apart from a few unusual circumstances, handling of the sysroots-components directory should be automatic, and recipes should not directly reference build/tmp/sysroots-components.

- **sysroots**: Previous versions of the OpenEmbedded build system used to create a global shared sysroot per machine along with a native sysroot.

- **stamps**: This directory holds information that BitBake uses for accounting purposes to track what tasks have run and when they have run. The directory is sub-divided by architecture, package name, and version.

- **work**: This directory contains architecture-specific work sub-directories for packages built by BitBake. All tasks are executed from the appropriate work directory. For example, the source for a particular package is unpacked, patched,

configured, and compiled all within its own work directory. Within the work directory, organization is based on the package group and version for which the source is being compiled as defined by the WORKDIR.

- **work-shared:** For efficiency, the OpenEmbedded build system creates and uses this

directory to hold recipes that share a work directory with other recipes. In practice, this is only used for gcc and its variants (e.g. gcc-cross, libgcc, gcc-runtime, and so forth).

14.4 Building Our First minimal image

There are a number of core- image recipes already available for you to use. These core recipes enable you to create a working linux image without doing absolutely any customization for your platform. A successful creation of such an image is proof that your build setup work sand you can then move on to doing more sophisticated builds for your own machines.

Some popular core images are:

- core-image-base
  – A console-only image that fully supports the target hardware.
- core-image-minimal
  – A small image allowing the device to just boot.
- core-image-sato
  – An image with Sato support, a mobile environment and visual style that works with mobile devices.
- core-image-clutter
  – An image with support for the OpenGL-based toolkit Clutter
- core-image-full-cmdline
  – A console-only image with more full-featured Linux system functionality installed.

For our first build we used **core-image-minimal**.

Once the build is complete, the output image can be found in tmp/deploy/images/qemux86-64 folder.

We can use hardware emulation to speed up the development process, as it enables a test run without involving any actual hardware.

QEMU is a free, open source software package that performs hardware virtualization.

QEMU-based machines allow testing and development without real hardware. ARMv5, ARMv7, ARMv8, MIPS, MIPS64, PowerPC, PowerPC 64, RISC-V 32, RISC-V 64, x86, and x86-64 emulations are currently supported.

OpenEmbedded Core provides the runqemu script tool, which is a wrapper to make use of
QEMU is easier. The way to run the script tool is as follows:
```
runqemu qemux86-64 nographic
```
The nographic argument tells runqemu that we are not interested in launching a GUI.

We can log in to the root account using an empty password. The system behaves as a regular machine, even when executed inside the QEMU. The process to deploy an image in real hardware varies, depending on the type of storage used, the bootloader, and so on.

14.5 Building Our Raspberry pi image

- Setting up build environment:
```
 source oe-init-build
```
- Bitbake layer configuration for Raspberry pi:
From the readme.md document of the meta-raspberrypi layer, we need at least meta-openembedded layer and probably some more depending on the features needed in the image.
The recipes within meta-openembedded enable builds for embedded architectures like ARM,MIPS, etc.
To add this board support to our project, we need to include the meta-raspberrypi meta layer, which is the BSP layer with support for the Raspberry Pi boards, including the Raspberry Pi4, but not limited to it.
For our build, we will also add other layers necessary for networking, python. We will first download these and then add to our build configuration:
```
git clone git://git.openembedded.org/meta-openembedded -b kirkstone
```

We then download the meta-raspberrypi layer source code. This layer contains all the recipes that are necessary to build basic images for the Raspberry Pi series of boards.
The meta-raspberrypi layer will contain the appropriate recipes for the kernel, drivers, configuration, etc. that will override those in the corresponding poky and meta-openembedded recipes. For example: the linux kernel configuration.
```
git clone git://git.yoctoproject.org/meta-raspberrypi -b kirkstone
```

Chapter 15:
Programming Enhancements

15.1 Why Choose C++?

C++ is a powerful and versatile programming language, a popular choice for developing high-performance systems and complex applications. Here are some reasons why it's a preferred option:

Exceptional Performance: C++ gives you direct control over hardware resources and memory, allowing for the creation of applications with extremely fast execution speeds. This is crucial in fields like game development, embedded systems, and real-time applications.

Precise Control: C++ enables low-level interaction with hardware, granting developers unparalleled ability to optimize performance and manage resources efficiently.

Flexibility and Multi-Paradigm: C++ supports Object-Oriented Programming (OOP), Generic Programming, and Procedural Programming, giving developers the flexibility to choose the most suitable approach for their project.

Stability and Reliability: Thanks to its long history and vast supporting community, C++ is known for its stability and ability to build robust and reliable systems.

Extensive Libraries: A massive number of libraries and tools are available for C++, simplifying the development process and providing ready-made solutions for many problems.

15.2 What is Qt?

Qt (pronounced "cute") is not just a library for creating graphical user interfaces (GUI); it's a comprehensive, cross-platform framework for application development. Qt is widely used to build rich and beautiful user interfaces, as well as applications that don't require a graphical interface.

15.3 Why Choose Qt?

While many frameworks are available for GUI development, Qt stands out in several key aspects:

Cross-Platform Development: This is Qt's most significant strength. You can write your code once and run it on various operating systems like Windows, macOS, Linux, Android, iOS, and even embedded systems, all while maintaining a native look and feel on each platform. This saves significant time and effort in development and maintenance.

Comprehensive Toolset: Qt isn't limited to just user interfaces. It provides modules for handling networking, databases, XML and JSON, 2D and 3D graphics, multithreading, and more. This means you often won't need to search for additional external libraries for most application needs.

Ease of Use and Learning: Despite its power, Qt is relatively easy to use, especially for developers familiar with object-oriented programming. It also offers visual design tools (like Qt Designer) to simplify interface building.

Large Community and Excellent Documentation: Qt boasts an active community, extensive learning resources, and comprehensive documentation, making it easy to get support and solve problems.

Performance: Being built on C++, Qt offers excellent performance for applications requiring intensive graphics processing or complex computations.

15.4 Why do we use C++ instead of C language?

1. Simplicity and Direct Memory Control

C is a simpler and more primitive language than C++. It doesn't have features like Classes, Inheritance, Templates, or Exception Handling. This simplicity means:

Easier to Learn: For beginners, C can be an easier starting point to grasp fundamental programming concepts and how programs interact with hardware.

Finer Memory Control: In C, the developer manually handles memory allocation and deallocation using functions like malloc() and free(). This gives absolute control over memory usage, which is vital in Embedded Systems or resource-constrained applications where every byte of memory matters. In C++, memory management can be more complex due to objects and smart pointers.

2.Embedded Systems and Kernel Development C is the preferred language for developing:

Operating Systems: The Linux kernel, for example, is almost entirely written in C. This is due to C's low-level memory control, high performance, and proximity to hardware.

Firmware and Microcontrollers: These environments are often extremely limited in terms of memory and processing power. C's compact and lightweight code is ideal for such applications.

Device Drivers: Developing device drivers requires precise control over hardware and direct memory access, which C excels at.

3.Absolute Performance and Compact Code While C++ can be very fast, sometimes:

C can produce more efficient and compact code: Because there are no "overheads" associated with C++ features like Virtual Tables or Run-Time Type Information (RTTI), the compiled C code can be smaller in size and faster in certain critical cases. This is beneficial in applications demanding maximum speed or a very small program footprint.

Direct Hardware Access: C naturally facilitates writing code that directly interacts with memory addresses and ports to manipulate hardware, a crucial advantage in system programming.

4.Compatibility and Easier Interfacing with Other Languages

Application Programming Interfaces (APIs): Many fundamental APIs and operating system interfaces (like the Win32 API) are designed with a C interface, making it easier and more natural to interface with them using C.

Use as a Lingua Franca: C is often used as an intermediate language to bridge libraries written in different languages, given its simple interfaces and high compatibility.

Chapter 16:
Future work and conclusion

16.1 Future Work

Building upon the robust foundation laid by this project, several avenues for future development can be explored to further enhance the capabilities and reach of the smart vehicle system:

Expansion of ADAS Features: Integrate and test additional Advanced Driver Assistance Systems beyond LDW and Night Vision. This could include Adaptive Cruise Control (ACC), Automatic Emergency Braking (AEB), Blind Spot Detection (BSD), and parking assistance systems. This would create a more comprehensive and robust ADAS suite.

Enhanced AI Capabilities: Further develop the AI Vision component by incorporating more advanced machine learning models for object recognition, prediction of pedestrian and vehicle movements, and even driver behavior analysis. This could lead to more nuanced and proactive safety interventions.
Predictive Maintenance through Logger Data: Leverage the extensive data collected by the logger system to implement predictive maintenance algorithms. By analyzing trends in system failures and anomalies, the system could forecast potential issues before they occur, allowing for scheduled maintenance and preventing breakdowns.

Over-the-Air (OTA) Updates for All ECUs: While FOTA is implemented for firmware, explore extending OTA capabilities to update software on all Electronic Control Units (ECUs) within the vehicle. This would ensure that the entire vehicle's software ecosystem can be kept up-to-date and secure without physical intervention.

Cybersecurity Enhancements: With increased connectivity and OTA updates, cybersecurity becomes paramount. Future work should focus on implementing advanced encryption, intrusion detection systems, and secure boot mechanisms to protect the vehicle from cyber threats and unauthorized access.

Integration with V2X Communication: Explore Vehicle-to-Everything (V2X) communication. This would allow the smart vehicle to communicate with other vehicles (V2V), infrastructure (V2I), and even pedestrians (V2P), enabling a more collaborative and safer driving environment by sharing real-time information on road conditions, hazards, and traffic.

Cloud-Based Diagnostics and Fleet Management: Further develop the cloud integration of the logger data to enable more sophisticated remote diagnostics and fleet management capabilities for commercial applications. This could include centralized monitoring, remote troubleshooting, and optimized maintenance schedules for a fleet of vehicles.

Human-Machine Interface (HMI) Refinements: Improve the driver feedback mechanisms and user interface for alerts and system status. This could involve more intuitive visual displays, haptic feedback, or even voice-guided alerts to ensure drivers receive critical information clearly and without distraction.

Real-World Testing and Validation: Beyond simulator testing with CARLA, plan for real-world testing of the developed system in controlled environments and eventually on public roads (with appropriate safety measures and regulations). This is crucial for validating the system's performance under diverse and unpredictable real-world conditions.

Energy Efficiency Optimization: As more electronic systems are integrated, optimizing their power consumption becomes important. Future work could focus on developing energy-efficient algorithms and hardware designs to minimize the impact on the vehicle's overall energy consumption.

Scalability and Modularity: Design the system with scalability and modularity in mind, allowing for easy integration of new features and hardware components in the future without a complete system overhaul.

16.2 Conclusion

This project, **SMART VEHICLE USING Logger,LDW, FOTA and AI,** addresses the growing complexity and potential failure risks of advanced electronic systems in modern vehicles. The core of the solution lies in developing a smart car equipped with a logger system, FOTA (Firmware Over-The-Air) technology, and AI Vision.

The logger system is a crucial component that continuously monitors and verifies the functionality of all critical vehicle systems before and during a journey. It detects and communicates potential issues in real-time to the driver, thereby reducing the risk of unexpected system failures and enhancing road safety. Detailed information about subsystem failures is logged for future diagnostics and analysis.

FOTA technology enables remote software updates, allowing for the proactive correction of software issues without the need for physical maintenance. This is particularly beneficial for automotive embedded systems with numerous Electronic Control Units (ECUs) that require frequent updates. The FOTA mechanism incorporates secure boot processes, ensuring software integrity through cryptographic signatures and hashing algorithms, and secure delivery infrastructure with authentication mechanisms.

AI Vision, specifically implemented through Lane Departure Warning (LDW) and Night Vision systems, significantly improves driver safety and comfort by enabling real-time decision-making. The Night Vision System leverages thermal or infrared cameras for enhanced visibility in low-light conditions, processing images with computer vision and AI techniques. LDW monitors lane markings and alerts the driver of unintended lane departures, with some systems even providing active steering intervention.

The project's hardware architecture incorporates an STM32F10x ARM controller for subsystem management and Raspberry Pi for AI processing, especially for vision-based systems. Communication between the main system and subsystems is managed via the CAN Bus. The development of a custom UDS (Unified Diagnostic Services) system was considered to manage costs and allow for customization specific to project needs,

focusing on essential services like reading and clearing Diagnostic Trouble Codes (DTCs).

Overall, this smart vehicle project provides a proactive approach to vehicle maintenance and safety, aiming to prevent unexpected breakdowns, ensure informed driver decisions, enhance road safety, and reduce maintenance costs.

References

[1] Ermagan, Vina, et al. "Towards model-based failure-management for automotive software." *Fourth International Workshop on Software Engineering for Automotive Systems (SEAS'07)*. IEEE, 2007.

[2] Hazen, Alyson, and John E. Ehiri. "Road traffic injuries: hidden epidemic in less developed countries." *Journal of the national medical association* 98.1 (2006): 73.

[3] Pohl, Jochen, Wolfgang Birk, and Lena Westervall. "A driver-distraction-based lane-keeping assistance system." *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering* 221.4 (2007): 541-552.

[4] Wood, Joanne M. "Nighttime driving: visual, lighting and visibility challenges." *Ophthalmic and physiological optics* 40.2 (2020): 187-201.

[5] Brown, Geoffrey. "Discovering the STM32 microcontroller." *Cortex* 3.34 (2012): 64.

[6] Molloy, Derek. Exploring Raspberry Pi: interfacing to the real world with embedded Linux. John Wiley & Sons, 2016.

[7] Gay, Warren. *Raspberry Pi hardware reference*. Apress, 2014.

[8] https://www.circuits-diy.com/hw201-infrared-ir-sensor-module/

[9] *Components101,* " Buzzer Pinout, Working, Specifications & Datasheet.", 25 Sept. 2017,

https://components101.com/misc/buzzer-pinout-working-datasheet.

[10] Matsuzaki, Takanori, et al. "Embedded controller based learning system for DC motor control." *Int J Innov Comput Inform Control* 15.3 (2019): 997-1007.

[11] H-Bridge Motor Driver – Working and Applications." *Electronics Tutorials*,

https://www.electronics-tutorials.ws/blog/h-bridge.html

[12] Raspberry Pi Camera Module with Night Vision – OV5647, 5MP, 160° FOV." *Waveshare*,

https://www.waveshare.com/wiki/RPi_IR-CUT_Camera

[13] *Components101, "* HC-05 Bluetooth Module – Pinout, Specifications & Working.", 15 Oct. 2018,

https://components101.com/wireless/hc-05-bluetooth-module

[14] Onwubolu, Godfrey C. *Introduction to SolidWorks: A comprehensive guide with applications in 3D printing*. CRC Press, 2017.

[15] Navet, Nicolas, and Françoise Simonot-Lion. "A review of embedded automotive protocols." *Automotive Embedded Systems Handbook* (2017): 4-1.

[16] *Texas Instruments*, "CAN Protocol: History, Functionality and Applications."

https://www.ti.com/interface/can/what-is-can.html

[17] Neto, Dario Almudi, and Luiz Eduardo Galvão Martins. "A requirements specification template of a communication network based on CAN protocol to automotive embedded systems." *Journal of Computer Science and Technology* 10.03 (2010): 143-149.

[18] Salunkhe, A. A., Pravin P. Kamble, and Rohit Jadhav. "Design and implementation of CAN bus protocol for monitoring vehicle parameters." *2016 IEEE international conference on recent trends in electronics, information & communication technology (RTEICT)*. IEEE, 2016.

[19] Ran, Li, et al. "Design method of CAN BUS network communication structure for electric vehicle." *International Forum on Strategic Technology 2010*. IEEE, 2010.

[20] *freedesktop.org*, *"What Is D- Bus?"* ,version 0.5.0, Red Hat Inc.et al.,

https://dbus.freedesktop.org/doc/dbus-tutorial.html

[21] freedesktop.org. "Getting Started with D- Bus: System vs Session Bus." *D-Bus Getting Started Guide*, freedesktop.org, 2025.

[22] "Introduction to D- Bus." *Developer Documentation*, KDE TechBase, 2025,

https://techbase.kde.org/Development/Tutorials/D-Bus/Creating_Interfaces

[23] Love, Robert. "Get on the D-BUS." *Linux Journal* 2005.130 (2005): 3.

[24] Salli, Olli. "Building object-oriented software with the D-Bus messaging system." (2012).

[25] Will, Newton C., et al. "A trusted message bus built on top of D-Bus." *Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg)*. SBC, 2020.

[26] Gruber, Nicole, et al. "Vision and night driving abilities of elderly drivers." *Traffic injury prevention* 14.5 (2013): 477-485.

[27] Vignesh, Ravikumar, and Prakash P. "Vision-Based Driver Assistance Systems Using Deep Learning and Night Vision Cameras: A Review." *Procedia Computer Science*, vol. 189, 2021, pp. 226–233. Elsevier

[28] Nieto, Marcos, et al. "Optimising computer vision based ADAS: vehicle detection case study." *IET Intelligent Transport Systems* 10.3 (2016): 157-164.

[29] Govardhan, P. *Night time pedestrian detection for Advanced Driving Assistance Systems (ADAS) using near infrared images*. Diss. 2014.

[30] Nain, Vidushi, et al. "A study on object detection using artificial intelligence and image processing–based methods." *Mathematical Models Using Artificial Intelligence for Surveillance Systems* (2024): 121-148.

[31] Nawaz, Saqib Ali, et al. "AI-based object detection latest trends in remote sensing, multimedia and agriculture applications." *Frontiers in Plant Science* 13 (2022): 1041514.

[32] Faisal, Mustafa M., et al. "Object detection and distance measurement using AI." *2021 14th International Conference on Developments in eSystems Engineering (DeSE)*. IEEE, 2021.

[33] Joshi, Prateek. *Artificial intelligence with python*. Packt Publishing Ltd, 2017.

[34] Chen, Chenyi, et al. "DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving." Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2015, pp. 2722–2730. IEEE

[35] Hassan, Walid, et al. "Real-Time Object Detection Using YOLOv4-Tiny on Raspberry Pi 4." International Journal of Advanced Computer Science and Applications, vol. 12, no. 10, 2021, pp. 645–652. https://thesai.org/Downloads/Volume12No10/Paper_85-Real_Time_Object_Detection.pdf

[36] Norris, Donald J. *Beginning artificial intelligence with the Raspberry Pi*. Vol. 363. Apress, 2017.

[37] Chien, Jong-Chih, Yong-Sheng Chen, and Jiann-Der Lee. "Improving night time driving safety using vision-based classification techniques." *Sensors* 17.10 (2017): 2199.

[38] Jiang, Zicong, et al. "Real-time object detection method based on improved YOLOv4-tiny." *arXiv preprint arXiv:2011.04244* (2020).

[39] Antony, Maria Merin, and Ruban Whenish. "Advanced driver assistance systems (ADAS)." Automotive Embedded Systems: Key Technologies, Innovations, and Applications. Cham: Springer International Publishing, 2021. 165-181.

[40] Lane, David, et al. *"Lane Departure Warning Systems and Lane Keeping Assist: A Review of the Literature." National Highway Traffic Safety Administration (NHTSA)*, 2020, https://www.nhtsa.gov/

[41] National Highway Traffic Safety Administration (NHTSA). *"Traffic Safety Facts 2022: Critical Reasons for Crashes Investigated in the National Motor Vehicle Crash Causation Survey."* U.S. Department of Transportation, 2023

[42] Raj, B., and R. Suresh. *"Design of Embedded Lane Detection System for Driver Assistance." International Journal of Engineering and Advanced Technology (IJEAT)*, vol. 8, no. 5, 2019, pp. 1321–1326.

[43] Manogaran, Gunasekaran, and Chiranjeevi Yelamarthi. *"Smart IR Sensor-Based Autonomous Car for Lane Detection and Obstacle Avoidance." Procedia Computer Science*, vol. 133, 2018, pp. 417–424.

[44] Hsiao, Pei-Yung, and Chun-Wei Yeh. "A portable real-time lane departure warning system based on embedded calculating technique." *2006 IEEE 63rd Vehicular Technology Conference*. Vol. 6. IEEE, 2006

[45] Sternlund, Simon, et al. "The effectiveness of lane departure warning systems—A reduction in real-world passenger car injury crashes." *Traffic injury prevention* 18.2 (2017): 225-229.

[46] Ahmed, S., and A. Farooq. *Firmware Over-the-Air (FOTA) Update Mechanisms in IoT Devices: A Comprehensive Review*. IEEE Access, vol. 9, 2021, pp. 144356–144375.

[47] Silicon Labs. *Understanding Firmware Over-The-Air (FOTA) Updates in Embedded Systems*. Application Note AN970, 2020.

[48] Tesla Motors. *Model S Software Update and Remote Bug Fixing via FOTA*. Tesla Blog Archives, 2016.

[49] IHS Automotive. *OEM Cost Savings and Market Projections with FOTA Updates*. IHS Market Research Report, 2015.

[50] Alhawari, Samer, et al. *Firmware Upgrade Strategies for Automotive ECUs in Connected Vehicles*. MDPI Electronics, vol. 9, no. 4, 2020.

[51] IEEE. Secure Firmware Update in IoT Devices: A Comparative Study. IEEE Internet of Things Journal, vol. 8, no. 3, 2021.

[52] Das, Saumik, and Subir Halder. Security Aspects of Over-The-Air Firmware Update in Embedded Systems. Springer, 2021.

[53] Kaur, Manpreet, and Harpreet Kaur. Reliable Update Mechanisms for Embedded Systems: A/B Partitioning and Rollback Strategies. International Journal of Computer Applications, vol. 180, no. 24, 2018.

[54] STMicroelectronics. STM32F1 Series Reference Manual. RM0008, Rev 21, 2022.

[55] STMicroelectronics. AN3155: USART Protocol Used in the STM32 Bootloader. Application Note, 2017.

[56] STMicroelectronics. ST-LINK/V2 In-Circuit Debugger/Programmer for STM32. User Manual UM1075, 2021.

[57] Mender.io. Mender Documentation: Overview and Architecture. Northern.tech, 2023.

[58] Mender.io. Best Practices for FOTA Security: TLS, Signing, and Rollback. Mender Whitepaper, 2022.

[59] Northern.tech. Mender Client Integration Guide for Raspberry Pi. Mender Documentation, 2023.

[60] Khan, Junaid, and F. Ahmed. *Designing Custom Bootloaders for STM32 Microcontrollers*. International Journal of Embedded Systems and Applications, vol. 10, no. 2, 2022.

[61] Vrachkov, Dimitar Georgiev, and Dimitar Georgiev Todorov. "Automotive diagnostic trouble code (DTC) handling over the Internet." 2018 IX National Conference with International Participation (ELECTRONICA). IEEE, 2018.

[62] Bengtner, Johannes, and Alex Smilevski. "Diagnostic Trouble Code Analysis." (2022).

[63] Zhang, Jie. Master of Science and Engineering. Diss. Nankai University, China, 2002, 2005.

[64] Summerfield, Mark. Advanced Qt programming: creating great software with C++ and Qt 4. Pearson Education, 2010.

[65] Sumorek, Andrzej, and Marcin Buczaj. "Problems of searching for failures and interpretation of error codes (DTCs) in modern vehicles." ECONTECHMOD: An International Quarterly Journal on Economics of Technology and Modelling Processes 5.3 (2016): 27-34.

[66] Hasan, Ragib. "History of linux." 2002,

[67] Economides, Nicholas, and Evangelos Katsamakas. "Linux vs. Windows: A comparison of application and platform innovation incentives for open source and proprietary software platforms." *The Economics of Open Source Software Development*. Elsevier, 2006. 207-218.

[68] Li, Yixiao, Yutaka Matsubara, and Hiroaki Takada. "A comparative analysis of RTOS and linux scalability on an embedded many-core processor." *Journal of Information Processing* 26 (2018): 225-236.

[69] Raghavan, Pichai, Amol Lad, and Sriram Neelakandan. *Embedded Linux system design and development*. Auerbach Publications, 2005.

[70] Profentzas, Christos, et al. "Performance of secure boot in embedded systems." *2019 15th International conference on distributed computing in sensor systems (DCOSS)*. IEEE, 2019.

[71] Lee, Chi-Tai, et al. "An application-oriented Linux kernel customization for embedded systems." *J. Inf. Sci. Eng.* 20.6 (2004): 1093-1107.

[72] Hollabaugh, Craig. *Embedded Linux: hardware, software, and interfacing*. Addison-Wesley Professional, 2002.

[73] Simmonds, Chris. *Mastering Embedded Linux Programming*. Packt Publishing Ltd, 2017.

[74] Sampath, Pradyumna, and B. Rachana Rao. "Efficient embedded software development using QEMU." *13th Real Time Linux Workshop*. 2011.

[75] Salvador, Otavio, and Daiane Angolini. *Embedded Linux Development with Yocto Project*. Packt Publishing Ltd, 2014.

[76] Salvador, Otavio, and Daiane Angolini. *Embedded Linux Development using Yocto Projects: Learn to leverage the power of Yocto Project to build efficient Linux-based products*. Packt Publishing Ltd, 2017.

[77] González, Alex. *Embedded Linux Development Using Yocto Project Cookbook: Practical recipes to help you leverage the power of Yocto to build exciting Linux-based systems*. Packt Publishing Ltd, 2018.

[78] Chatzigeorgiou, Alexander. "Performance and power evaluation of C++ object-oriented programming in embedded processors." Information and software Technology 45.4 (2003): 195-201.

[79] Morris, Ezra. The Design and Deployment of Cross-Platform User Interface for the Analysis, Control and Management of Disparate Embedded Systems. Diss. University of Huddersfield, 2015.

[80] Barr, Michael. Programming embedded systems in C and C++. " O'Reilly Media, Inc.", 1999.

ثالثاً: الذكاء الاصطناعي

في أنظمة مساعدة السائق يركز المشروع على دمج الذكاء الاصطناعي في نظامين رئيسيين لمساعدة السائق

نظام الرؤية الليلية **(Night Vision System):** مصمم لتوفير رؤية محسنة في ظروف الإضاءة المنخفضة أو الظلام التام. يستخدم نظام الرؤية الليلية كاميرات حرارية أو الأشعة تحت الحمراء وجهاز للحصول على بيانات المسافة ثلاثية الأبعاد والكشف عن الأجسام.

نظام التحذير من مغادرة المسار: (**Lane Departure Warning – LDW**) ميزة حاسمة لتعزيز السلامة من خلال مساعدة السائقين على البقاء داخل مساراتهم. باستخدام حساسات تتبع الخطوط، يراقب نظام علامات المسار على الطريق. إذا اكتشف النظام مغادرة غير مقصودة للمسار دون إشارة، فإنه يوفر تنبيهات مرئية أو سمعية للسائق

رابعا: البنية التحتية والمكونات

تعتمد البنية الصلبة على عدة وحدات تحكم دقيقة، مثل متحكمات **STM32F103C8T6 ARM** لإدارة الأنظمة الفرعية، و **Raspberry Pi** لمعالجة الذكاء الاصطناعي، خاصة لأنظمة الرؤية. يتم استخدام وحدة اتصالات عن بعد بين النظام الرئيسي والأنظمة الفرعية عبر بروتوكول **CANBus** لضمان تبادل بيانات موثوق به .

تم بحمد الله

الملخص بالعربية

في ظل التطور المتسارع لأنظمة المركبات الحديثة، أصبحت الاعتمادية والذكاء محركين أساسيين للابتكار في قطاع السيارات. مع ازدياد تعقيد الأنظمة الإلكترونية المتقدمة التي تعزز السلامة والراحة والأداء، تتصاعد مخاطر الأعطال الحرجة التي قد تحدث دون سابق إنذار للسائق. يعتمد السائقون بشكل كبير على هذه الأنظمة لضمان قيادة آمنة وفعالة، وغالبًا ما يظلون غير مدركين لأي أعطال حتى فوات الأوان. من هذا المنطلق، يأتي مقترح مشروع "المركبة الذكية باستخدام المسجل (Logger)، تحديث البرامج عبر الهواء (FOTA)، والذكاء الاصطناعي (AI)" ونظام التحذير من مغادرة المسار ليقدم حلاً شاملاً لهذه التحديات.

يهدف هذا المشروع إلى تطوير مركبة قادرة على المراقبة الاستباقية لأنظمتها، وتلقي التحديثات البرمجية عن بُعد، والاستفادة من قدرات الذكاء الاصطناعي لتحسين السلامة وتجربة القيادة بشكل عام، مما يقلل من الأعطال غير المتوقعة ويزيد من وعي السائق بحالة مركبته في جميع الأوقات.

أولا: نظام المسجل

يعتبر نظام المسجل مكونًا حيويًا للمراقبة والتحقق من وظائف جميع أنظمة المركبة الحيوية قبل بدء الرحلة. يقوم المسجل بالكشف عن أي مشكلات محتملة وإبلاغ السائق بها في الوقت الفعلي، مما يقلل من مخاطر الأعطال غير المتوقعة أثناء القيادة. يتم تسجيل بيانات تفصيلية حول أعطال الأنظمة الفرعية في قاعدة بيانات داخلية للاستفادة منها في التشخيص المستقبلي وتحليل الاتجاهات. يمكن أيضًا تحميل هذه البيانات بشكل دوري إلى التخزين السحابي للمراقبة عن بعد وإجراء التحديثات عبر الهواء. يشمل ذلك مسؤوليات تشخيصية الأنظمة مثل نظام التحذير من مغادرة المسار ونظام الرؤية الليلية

ثانياً: تحديث البرامج عبر الهواء

تتيح تقنية تحديث البرامج عبر الهواء للمركبة استقبال تحديثات البرامج عن بعد، مما يسمح بمعالجة مشكلات البرامج دون الحاجة إلى صيانة مادية.
تحديث البرامج عبر الهواء مفيدة بشكل خاص للأنظمة المدمجة في السيارات التي تحتوي على عدد كبير من وحدات التحكم الإلكترونية والتي تتطلب تحديثات متكررة. تتكون تحديث البرامج عبر الهواء من بنية تحتية آمنة تتضمن آليات مصادقة قوية، إصلاح الأخطاء، وتحسين الأداء، وإضافة ميزات جديدة